

การจัดเรียงข้อมูลแบบขนานโดยใช้หลายสแต็กและการออปติไมเซชัน

MULTISTACK PARALLEL SORTING ALGORITHM AND ITS OPTIMIZATION



อภิสิทธิ์ รัตนตรานุรักษ์

APISIT RATTANATRANURAK

วิทยานิพนธ์นี้สำหรับการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรดุษฎีบัณฑิต

สาขาวิชาวิศวกรรมไฟฟ้า

คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

พ.ศ. 2564

KMITL-2021-EN-D-018-019

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

MULTISTACK PARALLEL SORTING ALGORITHM AND ITS OPTIMIZATION

APISIT RATTANATRANURAK

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENT FOR THE DEGREE OF
DOCTOR OF ENGINEERING IN ELECTRICAL ENGINEERING
SCHOOL OF ENGINEERING
KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG
2021

KMITL-2021-EN-D-018-019

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้



COPYRIGHT 2021

SCHOOL OF ENGINEERING

KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

หัวข้อวิทยานิพนธ์	การจัดเรียงข้อมูลแบบขนานโดยใช้หลายสแต็กและการออปติไมเซชัน
นักศึกษา	นายอภิสิทธิ์ รัตนตรานุรักษ์
รหัสประจำตัว	57601413
ปริญญา	วิศวกรรมศาสตรดุษฎีบัณฑิต
สาขาวิชา	วิศวกรรมไฟฟ้า
พ.ศ.	2564
อาจารย์ที่ปรึกษาวิทยานิพนธ์	ผศ.ดร.สุรินทร์ กิตติธรรมกุล

บทคัดย่อ

วิทยานิพนธ์ฉบับนี้นำเสนออัลกอริธึมการจัดเรียงข้อมูลแบบขนานด้วยเทคนิคมัลติเธรดบนซีพียูชนิดมัลติคอร์ (Multicore) และแมนีคอร์ (Many-Core) อัลกอริธึมมีการทำงานโดยแบ่งข้อมูล (Partition) แบบขนานโดยใช้ไพวอท 1 ค่า (Single-Pivot Value) และสามารถแบ่งอาเรย์ข้อมูลย่อยซ้ำได้อีกจนกระทั่งอาเรย์ย่อยมีขนาดสั้นพอที่จะนำไปจัดเรียงข้อมูลด้วยเธรดเดียวโดยอิสระ

การแบ่งข้อมูลในอาเรย์ย่อยแต่ละรอบจะสร้างเธรดจำนวนหนึ่งแต่ละเธรดมีสแต็กซ้ายและสแต็กขวาของตนเองโดยจะวางขอบเขตของบล็อกข้อมูลในสแต็กแบบ Last-In First-Out เพื่อทำการเปรียบเทียบ/สวอป (Compare/Swap) ข้อมูลภายในบล็อกซ้ายและบล็อกขวานสแต็กด้านใดด้านหนึ่งว่างลง เธรดทั้งหมดจะหยุดชั่วคราวเพื่อซิงโครไนซ์ แล้วจึงแบ่งข้อมูลแบบอนุกรมจนสำเร็จ อัลกอริธึมมีประสิทธิภาพค่อนข้างดีบนเครื่องคอมพิวเตอร์ที่หลากหลายรูปแบบได้แก่ Intel i7-2600, Intel X5670, AMD R7-1700 และ AMD R9-2920

การออปติไมซ์อัลกอริธึมเพื่อเพิ่มประสิทธิภาพ ได้แก่ การปรับขนาดบล็อก การแก้ไขปัญหาค่าไพวอทเอียง การปรับอัตราส่วนเธรดในการแบ่งข้อมูล การจัดวางเธรดแบบพลวัต อัลกอริธึม Depth First Traversal และการตั้งจำนวนเธรดต่ำสุด เป็นต้น แล้วจึงทดสอบอัลกอริธึมที่ออปติไมซ์แล้วเปรียบเทียบกับประสิทธิภาพกับ Balanced QuickSort และ Multiway Merge sort ซึ่งเป็นอัลกอริธึมการจัดเรียงข้อมูลแบบมาตรฐานแบบขนานเช่นเดิมบนซีพียูขนาดเล็ก ARM Cortex A72 และขนาดใหญ่ AMD R9-2920 และได้ผลที่น่าประทับใจในแง่ของเวลาเฉลี่ยและเสถียรภาพของเวลาที่ใช้ในการจัดเรียงข้อมูล การใช้งานซีพียู และหน่วยความจำ (DRAM)

Thesis	MultiStack Parallel Sorting Algorithm and Its Optimization
Student	Mr.Apisit Rattanatanurak
Student ID.	57601413
Degree	Doctor of Engineering
Program	Electrical Engineering
Year	2021
Thesis Advisor	Asst.Prof.Dr.Surin Kittitornkun

ABSTRACT

This thesis proposes a multithreaded MultiStack Parallel Sorting (MSPSort) Algorithm and its optimization on multicore CPUs. It recursively partitions an unsorted array with a single-pivot value in parallel. The (sub)array partitioning continues until it can yield short enough subarrays to be sorted independently as a thread.

Each level of subarray partition is a multithreaded function. Each thread has its own left and right stacks to hold data block boundaries. Data elements within left and right blocks are compared/swapped according to the single-pivot value until either left or right stack is empty. All threads must be synchronized as a barrier here. Then, the serial partition can cleanup the rest. MSPSort can perform well on a variety of multicore CPUs including Intel i7-2600, Intel X5670, AMD R7-1700, AMD R9-2920.

The MultiStack Partition algorithm can be optimized further to improve its performance, for instance, Block size, Handling pivot skew value, Partition Thread Ratio, Dynamic Thread Allocation, Depth First Traversal algorithm, and Minimum Threads. The optimized MSPSort is benchmarked again with the standard Balanced QuickSort and MultiWay merge sort algorithms on ARM Cortex A72 and AMD R9-2920 CPUs. MSPSort can still perform well in various aspects including run time on average/stability, CPU and Main memory (DRAM) Utilization.

กิตติกรรมประกาศ

วิทยานิพนธ์เล่มนี้สำเร็จได้ด้วยความกรุณาจากอาจารย์ที่ปรึกษาผู้ช่วยศาสตราจารย์ ดร. สุรินทร์ กิตติธรรมกุล ที่ให้ความช่วยเหลือ ให้คำชี้แนะช่วยแก้ปัญหาตลอดจนให้ความรู้และประสบการณ์ที่ดีแก่ข้าพเจ้า

ขอขอบพระคุณคณะกรรมการสอบวิทยานิพนธ์ที่ได้กรุณาให้คำแนะนำตลอดจนข้อชี้แนะจนในที่สุดทำให้วิทยานิพนธ์ฉบับนี้สำเร็จลงได้

ขอขอบพระคุณบิดามารดา ที่คอยเป็นที่ปรึกษา และเป็นกำลังใจ ระหว่างการทำวิทยานิพนธ์ฉบับนี้

ขอขอบคุณผู้ช่วยศาสตราจารย์ ดร. ศิริลักษณ์ เกตุฉาย ซึ่งให้คำแนะนำในการปฏิบัติงาน การแบ่งเวลา หรือกระทั่งการทำงานบริหาร พร้อมทั้งศึกษาต่อ จนกระทั่งทำให้วิทยานิพนธ์นี้สำเร็จลงได้

ขอขอบพระคุณเพื่อนร่วมงาน คณาจารย์ เจ้าหน้าที่ ในมหาวิทยาลัยราชภัฏสวนสุนันทาที่ได้ให้ความช่วยเหลือสนับสนุนคอยให้คำปรึกษาเรื่องอุปกรณ์เพื่อใช้ในการทดลองภายในงานวิทยานิพนธ์ฉบับนี้

สำหรับคุณงามความดีอันใดที่เกิดจากวิทยานิพนธ์ฉบับนี้ ข้าพเจ้าขอมอบให้กับบิดามารดา ซึ่งเป็นที่รักและเคารพยิ่ง ตลอดจนครูอาจารย์ที่เคารพทุกท่านที่ได้ประสิทธิ์ประสาทวิชาความรู้และถ่ายทอดประสบการณ์ที่ดีให้แก่ข้าพเจ้า

อภิสิทธิ์ รัตนตรานุรักษ์

สารบัญ

	หน้า
บทคัดย่อภาษาไทย	I
บทคัดย่อภาษาอังกฤษ.....	II
กิตติกรรมประกาศ.....	III
สารบัญ.....	IV
สารบัญตาราง.....	VI
สารบัญภาพ.....	IX
บทที่ 1 บทนำ.....	1
1.1 ความเป็นมา.....	1
1.2 ความมุ่งหมายและวัตถุประสงค์.....	2
1.3 ประโยชน์ที่คาดว่าจะได้รับ.....	2
1.4 ขอบเขตของวิทยานิพนธ์.....	2
1.5 ขั้นตอนการดำเนินงาน.....	3
บทที่ 2 บททบทวนวรรณกรรม.....	4
2.1 การจัดเรียงข้อมูลโดยใช้วิธีการดีไวน์และคองเคอะแบบขนาน.....	4
2.2 STLSort โหมดการทำงานแบบตามลำดับและขนาน.....	8
2.2.1 Balanced QuickSort.....	8
2.2.2 MultiWay Merge Sort.....	8
2.3 โอเพนเอ็มพี.....	9
บทที่ 3 การจัดเรียงแบบขนานโดยใช้หลายสแต็ก.....	12
3.1 Recursive MultiStack Parallel Partition Phase.....	12
3.2 Sorting Phase.....	13
3.3 BF-DF Scheduling Algorithms.....	14
3.3.1 RAL vs LAL.....	14
3.3.2 SPF vs LPF.....	15
3.4 การทดลอง ผลลัพธ์ และอภิปรายผลการทดลอง.....	19
3.4.1 สภาพแวดล้อมในการทดลอง.....	19
3.4.2 การวัดประสิทธิภาพ.....	21
3.5 ผลการทดลองและการอภิปรายผลการทดลอง.....	23
3.5.1 การทดลอง MSPPartition ในการแบ่งข้อมูลครั้งแรก.....	23
3.5.2 การทดลองจัดเรียงข้อมูลแบบอิสระกันของแต่ละบล็อก.....	26

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

สารบัญ (ต่อ)

	หน้า
3.5.3 MSPSort โดยจัดงานแบบ BF.....	28
3.5.4 MSPSort โดยจัดงานแบบ BF-DF.....	29
บทที่ 4 การออกแบบอัลกอริธึมการจัดเรียงแบบขนานโดยใช้หลายสแต็ก.....	32
4.1 ขนาดบล็อก B.....	32
4.2 การแก้ไขปัญหาไฟวอทเอียงใน MSPPartition.....	34
4.3 อัตราส่วนเธรดในการแบ่งข้อมูล θ	36
4.4 การจัดวางเธรดแบบพลวัต.....	37
4.5 อัลกอริธึม Depth First Traversal.....	37
4.6 ค่าเธรดต่ำสุด $\tau_{\min}$	39
4.7 การทดลอง และตัวชี้วัด.....	40
4.7.1 เครื่องที่ใช้ในการทดลอง.....	40
4.7.2 ตัวชี้วัดในการทดลอง.....	41
4.8 ผลการทดลองและอภิปรายผลการทดลอง.....	43
4.8.1 BF-MSPSort และค่าขนาดบล็อก B.....	43
4.8.2 BF-MSPSort และค่าขนาด U_{sort} , m , $\tau_{\min}$	45
4.8.3 BF-MSPSort เปรียบเทียบกับการจัดลำดับงานแบบ BF-DF.....	45
4.8.4 RAL-MSPSort กับอัตราส่วนเธรดในการแบ่งข้อมูล θ	46
4.8.5 RAL-MSPSort กับค่าลดทอนเธรด α	48
4.8.6 RAL-MSPSort กับค่าวิคคูณเธรด β	50
4.9 การเปรียบเทียบเวลาต่อข้อมูล 1 ล้านตัวต่อแกนประมวลผลกับอัลกอริธึมการจัดเรียงข้อมูลอื่น.....	53
บทที่ 5 บทสรุป.....	54
5.1 สรุปผลการวิจัย.....	54
5.2 ข้อเสนอแนะ.....	55
5.3 แนวทางการพัฒนาต่อ.....	56
เอกสารอ้างอิง.....	57
ภาคผนวก	59

สารบัญตาราง

ตารางที่	หน้า
2.1 การเปรียบเทียบการจัดเรียงข้อมูลแบบขนานในเชิงของวิธีการแบ่งข้อมูล คอขวด การทำรีเคอร์ซีฟ เวลา และไลบรารี ที่ใช้.....	6
3.1 เครื่องที่ใช้ในการทดลอง.....	20
3.2 ตัวแปรที่ใช้ในการทดลอง MSPSort.....	21
3.3 คู่ m และ r 3 อันดับแรก ในการจัดงานแบบ BF ในทุกขนาดข้อมูลที่ $B=0.01$ $U_{stl} = 0.5, 1, 2, 4$ โดยทดสอบ 20 รอบ.....	29
3.4 พารามิเตอร์ที่เลือกใช้ในการทดลอง U_{stl} : U_{df} : m : r เมื่อ $B=0.01$ จากการทดลอง 100 ครั้ง.....	30
4.1 ข้อมูลของเครื่องที่ใช้ในการทดลอง.....	40
4.2 พารามิเตอร์ที่ใช้ในการทดลองของ MSPSort, BF: Bread-First, DF: Depth-First.....	41
4.3 $\bar{T}_{10}$ และ σ_{10} ของ BF-MSPSort ด้วยชนิดตัวแปร Uint32 ในแต่ละขนาดบล็อก B บน Pi4 จำนวน 100 ครั้ง.....	43
4.4 $\bar{T}_{10}$ และ σ_{10} ของ BF-MSPSort ด้วยชนิดตัวแปร Double ในแต่ละขนาดบล็อก B บน Pi4 จำนวน 100 ครั้ง.....	44
4.5 $\bar{T}_{100}$ และ σ_{100} ของ BF-MSPSort ด้วยชนิดตัวแปร Uint32 ในแต่ละขนาดบล็อก B บน R9-2920 จำนวน 100 ครั้ง.....	44
4.6 $\bar{T}_{100}$ และ σ_{100} ของ BF-MSPSort ด้วยชนิดตัวแปร Double ในแต่ละขนาดบล็อก B บน R9-2920 จำนวน 100 ครั้ง.....	45
4.7 ค่าพารามิเตอร์ (B , U_{sort} , m , τ_{min}) ที่ดีที่สุดของการจัดงานแบบ BF ในทุก N จำนวน 20 ครั้ง.....	45
4.8 ค่าพารามิเตอร์ที่ถูกเลือก (B , U_{sort} , U_{trav} , m , τ_{min}) ที่ดีที่สุดของการจัดงานแบบ RAL-MSPSort ในทุก N จำนวน 100 ครั้ง.....	46
4.9 ค่า $\bar{T}$ และ σ_T ของ RAL-MSPSort ชนิดข้อมูล Uint32 และ Double ในแต่ละค่า N และค่าอัตราส่วนเรต θ บนเครื่อง Pi4 จำนวน 100 ครั้ง.....	47
4.10 ค่า $\bar{T}$ และ σ_T ของ RAL-MSPSort ชนิดข้อมูล Uint32 และ Double ในแต่ละค่า N และค่าอัตราส่วนเรต θ บนเครื่อง R9-2920 จำนวน 100 ครั้ง.....	48

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

สารบัญตาราง (ต่อ)

ตารางที่	หน้า
4.11 ค่า $\bar{T}$ และ σ_T ของ RAL-MSPSort ชนิดข้อมูล Uint32 และ Double ที่ $\theta = 1.00$ ในแต่ละค่า N , α และ β บนเครื่อง Pi4 จำนวน 100 ครั้ง.....	49
4.12 ค่า $\bar{T}$ และ σ_T ของ RAL-MSPSort ชนิดข้อมูล Uint32 และ Double ที่ $\theta = 1.00$ ในแต่ละค่า N , α และ β บนเครื่อง R9-2920 จำนวน 100 ครั้ง.....	50
4.13 เวลาในการจัดเรียงข้อมูลแบบขนานต่อล้านข้อมูลต่อแกนประมวลผลเมื่อเปรียบ เทียบกับอัลกอริธึมอื่น.....	53
ก.1 เวลาที่ในการจัดเรียง $\bar{T}$ ของ MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิด ข้อมูล Uint32 ตามขนาด N บนเครื่อง i7-2600 โดยทดลอง 1000 ครั้ง.....	59
ก.2 เวลาที่ในการจัดเรียง $\bar{T}$ ของ MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิด ข้อมูล Uint64 ตามขนาด N บนเครื่อง i7-2600 โดยทดลอง 1000 ครั้ง.....	60
ก.3 เวลาที่ในการจัดเรียง $\bar{T}$ ของ MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิด ข้อมูล Double ตามขนาด N บนเครื่อง i7-2600 โดยทดลอง 1000 ครั้ง.....	61
ก.4 เวลาที่ในการจัดเรียง $\bar{T}$ ของ MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิด ข้อมูล Uint32 ตามขนาด N บนเครื่อง R7-1700 โดยทดลอง 1000 ครั้ง.....	62
ก.5 เวลาที่ในการจัดเรียง $\bar{T}$ ของ MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิด ข้อมูล Uint64 ตามขนาด N บนเครื่อง R7-1700 โดยทดลอง 1000 ครั้ง.....	63
ก.6 เวลาที่ในการจัดเรียง $\bar{T}$ ของ MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิด ข้อมูล Double ตามขนาด N บนเครื่อง R7-1700 โดยทดลอง 1000 ครั้ง.....	64
ก.7 เวลาที่ในการจัดเรียง $\bar{T}$ ของ MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิด ข้อมูล Uint32 ตามขนาด N บนเครื่อง X5670 โดยทดลอง 1000 ครั้ง.....	65
ก.8 เวลาที่ในการจัดเรียง $\bar{T}$ ของ MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิด ข้อมูล Uint64 ตามขนาด N บนเครื่อง X5670 โดยทดลอง 1000 ครั้ง.....	66
ก.9 เวลาที่ในการจัดเรียง $\bar{T}$ ของ MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิด ข้อมูล Double ตามขนาด N บนเครื่อง X5670 โดยทดลอง 1000 ครั้ง.....	67
ก.10 เวลาที่ในการจัดเรียง $\bar{T}$ ของ MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิดข้อมูล Uint32 ตามขนาด N บนเครื่อง R9-2920 โดยทดลอง 1000 ครั้ง.....	68
ก.11 เวลาที่ในการจัดเรียง $\bar{T}$ ของ MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิดข้อมูล Uint64 ตามขนาด N บนเครื่อง R9-2920 โดยทดลอง 1000 ครั้ง.....	69
ก.12 เวลาที่ในการจัดเรียง $\bar{T}$ ของ MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิดข้อมูล Double ตามขนาด N บนเครื่อง R9-2920 โดยทดลอง 1000 ครั้ง.....	70

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า

ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

สารบัญตาราง (ต่อ)

ตารางที่	หน้า
ข.1 ตัวชี้วัดของ RAL-MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิดข้อมูล Uint32 ที่ $\theta = 1.00$ ในแต่ละ N บนเครื่อง Pi4 จำนวน 1000 ครั้ง เมื่อ $B=128KB$, $U_{sort} = 8MB$, $m=1$, $\tau_{min}=2$, $\alpha = 2$, $\beta=OFF$	71
ข.2 ตัวชี้วัดของ RAL-MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิดข้อมูล Double ที่ $\theta = 1.00$ ในแต่ละ N บนเครื่อง Pi4 จำนวน 1000 ครั้ง เมื่อ $B=128KB$, $U_{sort} = 8MB$, $m=1$, $\tau_{min}=2$, $\alpha = 2$, $\beta=OFF$	72
ข.3 ตัวชี้วัดของ RAL-MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิดข้อมูล Uint32 ที่ $\theta = 1.00$ ในแต่ละ N บนเครื่อง R9-2920 จำนวน 1000 ครั้ง เมื่อ $B=128KB$, $U_{sort} = 8MB$, $m=1$, $\tau_{min}=2$, $\alpha = 2$, $\beta=OFF$	73
ข.4 ตัวชี้วัดของ RAL-MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิดข้อมูล Double ที่ $\theta = 1.00$ ในแต่ละ N บนเครื่อง R9-2920 จำนวน 1000 ครั้ง เมื่อ $B=128KB$, $U_{sort} = 8MB$, $m=1$, $\tau_{min}=2$, $\alpha = 2$, $\beta=OFF$	74

สารบัญภาพ

ภาพที่	หน้า
2.1 ฟังก์ชัน/จอย โมเดลในโอเพนเอ็มพี.....	10
2.2 การฟังก์ชัน/จอย โมเดล แบบซ้อน ในโอเพนเอ็มพี.....	10
3.1 $\bar{T}_{100M}$ (กราฟแท่ง) และ $\pm\sigma_{100M}$ (กราฟความผิดพลาด) ของการแบ่งข้อมูลรอบเป็น แรก ที่ $B=\{10^{-4}, 0.001, 0.01, 0.1, 1\} \times K_{13}$ ที่ $m=1$ บนเครื่อง i7-2600 เมื่อข้อมูล เป็น Uint32.....	24
3.2 $\bar{T}_{100M}$ (กราฟแท่ง) และ $\pm\sigma_{100M}$ (กราฟความผิดพลาด) ของการแบ่งข้อมูลรอบ แรก ที่ $B=\{10^{-4}, 0.001, 0.01, 0.1, 1\} \times K_{13}$ ที่ $m=1$ บนเครื่อง X5670 เมื่อข้อมูล เป็น Uint32.....	24
3.3 $\bar{T}_{100M}$ (กราฟแท่ง) และ $\pm\sigma_{100M}$ (กราฟความผิดพลาด) ของการแบ่งข้อมูลรอบแรก ที่ $B=\{10^{-4}, 0.001, 0.01, 0.1, 1\} \times K_{13}$ ที่ $m=1$ บนเครื่อง R7-1700 เมื่อข้อมูลเป็น Uint32.....	25
3.4 $\bar{T}_{100M}$ (กราฟแท่ง) และ $\pm\sigma_{100M}$ (กราฟความผิดพลาด) ของการแบ่งข้อมูลรอบ แรก ที่ $B=\{10^{-4}, 0.001, 0.01, 0.1, 1\} \times K_{13}$ ที่ $m=1$ บนเครื่อง R9-2920 เมื่อ ข้อมูลเป็น Uint32.....	25
3.5 $\bar{T}_{100M}$ (กราฟแท่ง) และ σ_{100M} (กราฟความผิดพลาด) ของจัดเรียงข้อมูลแบบอิสระ จากกัน ที่ $B=\{10^{-4}, 0.001, 0.01, 0.1, 1\} \times K_{13}$ ที่ $m=1$ บนเครื่อง i7-2600 เมื่อ ข้อมูลเป็น Uint32.....	26
3.6 $\bar{T}_{100M}$ (กราฟแท่ง) และ σ_{100M} (กราฟความผิดพลาด) ของจัดเรียงข้อมูลแบบอิสระ จากกัน ที่ $B=\{10^{-4}, 0.001, 0.01, 0.1, 1\} \times K_{13}$ ที่ $m=1$ บนเครื่อง X5670 เมื่อ ข้อมูลเป็น Uint32.....	27
3.7 $\bar{T}_{100M}$ (กราฟแท่ง) และ σ_{100M} (กราฟความผิดพลาด) ของจัดเรียงข้อมูลแบบอิสระ จากกัน ที่ $B=\{10^{-4}, 0.001, 0.01, 0.1, 1\} \times K_{13}$ ที่ $m=1$ บนเครื่อง R7-1700 เมื่อ ข้อมูลเป็น Uint32.....	27
3.8 $\bar{T}_{100M}$ (กราฟแท่ง) และ σ_{100M} (กราฟความผิดพลาด) ของจัดเรียงข้อมูลแบบอิสระ จากกัน ที่ $B=\{10^{-4}, 0.001, 0.01, 0.1, 1\} \times K_{13}$ ที่ $m=1$ บนเครื่อง R9-2920 เมื่อ ข้อมูลเป็น Uint32.....	28
4.1 $\bar{T}/N$ ของ RAL-MSPSort ชนิดข้อมูล Uint32 บนเครื่อง R9-2920 ที่ $B=128kB$ $U_{sort}=8MB$, $\alpha=2$ ในแต่ละค่า N จำนวน 1000 ครั้ง.....	52

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

สารบัญภาพ (ต่อ)

ภาพที่	หน้า
4.2 $\bar{T}/N$ ของ RAL-MSPSort ชนิดข้อมูล Double บนเครื่อง R9-2920 ที่ $B=128kB$ $U_{\text{sort}}=8MB$, $\alpha=2$ ในแต่ละค่า N จำนวน 1000 ครั้ง.....	52
ค.1 ตัวอย่างการเกิดไฟวอทเอียงใน MSPPartition.....	76



เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
 ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

บทที่ 1

บทนำ

1.1 ความเป็นมา

ทุกวันนี้มีการใช้งานซีพียูที่มีหลายแกนทั้งในเครื่องส่วนบุคคล เครื่องแม่ข่าย ทั้งในรูปแบบการเข้าถึงหน่วยความจำแบบยูนิฟอร์มหรือนอน-ยูนิฟอร์ม (Uniform/Non-Uniform Memory Access: UMA/NUMA) ซึ่งเทคโนโลยีในปัจจุบันมีการใช้สมาร์ทโฟน แท็บเล็ต และอุปกรณ์พกพาอื่นมากยิ่งขึ้น อุปกรณ์เหล่านี้มีการติดตั้งซีพียูหลายแกนเพื่อให้สามารถทำงานบนแอปพลิเคชันที่มีความซับซ้อนในการประมวลผล ดังนั้นเมื่อมีการติดตั้งซีพียูหลายแกนบนอุปกรณ์เหล่านี้ อัลกอริธึมที่ใช้งานบนอุปกรณ์ดังกล่าว จะต้องมีการทำงานแบบขนานเพื่อให้มีการนำเอาทรัพยากรภายในอุปกรณ์พกพามาใช้ให้มีประสิทธิภาพสูงสุด

การจัดเรียงข้อมูลและการแบ่งข้อมูล (Partitioning) เป็นอัลกอริธึมพื้นฐานที่เป็นที่รู้จัก ซึ่งมักจะมีการใส่ไว้ในโปรแกรมหรือแอปพลิเคชันต่างๆ อัลกอริธึมการจัดเรียงข้อมูล มีอยู่หลากหลายรูปแบบ อย่างไรก็ตามรูปแบบที่เป็นที่รู้จักคือควิกซอร์ต (Quicksort) อัลกอริธึมนี้มีรูปแบบขั้นตอนวิธีแบ่งแยกและเอาชนะ (Divide & conquer: D&C) ซึ่งมีหลักการคือแบ่งข้อมูลก่อน (Partitioning) โดยส่วนใหญ่จะมีวิธีการแบ่งข้อมูลที่เป็นที่นิยมคือ Hoare's [1] และ Lomuto's [2] ถัดจากนั้นจะเป็นการจัดเรียงข้อมูล โดยวิทยานิพนธ์ฉบับนี้จะนำเสนอการจัดเรียงข้อมูลแบบขนานโดยแก้ปัญหาเรื่องการเข้าถึงหน่วยความจำที่มีจำนวนมากขณะที่มีการเปรียบเทียบและสลับ (Compare & Swap) ซึ่งเป็นปัญหาใหญ่ปัญหาหนึ่งในการจัดเรียงข้อมูลขนาดใหญ่

ในวิทยานิพนธ์นี้จะมีการนำเสนออัลกอริธึมการแบ่งข้อมูลแบบ 1 ไพวอต (Pivot) ในรูปแบบบล็อก ที่ชื่อว่า **MultiStack Parallel Partition (MSPPartition)** ซึ่งจะถูกนำมาประยุกต์ใช้ในการจัดเรียงข้อมูลที่มีพื้นฐานอยู่บนควิกซอร์ต ที่ชื่อว่า **MSPSort** โดยอัลกอริธึมการจัดเรียงข้อมูลนี้จะมีการแบ่งข้อมูลและรีเคอร์ซีฟจนกระทั่งความยาวของอาร์เรย์มีขนาดสั้นเพียงพอ จึงนำไปจัดเรียงข้อมูลสิ่งที่แตกต่างจากอัลกอริธึมอื่นที่เป็นแบบบล็อกคือ อัลกอริธึมนี้ใช้โครงสร้างข้อมูลสแต็กเข้ามาช่วยในการเก็บขอบเขตบล็อกแทนการใช้คิว (queue) หรือดีค (deque) ซึ่งอัลกอริธึม MSPSort จะมีรูปแบบอิน-เพลส (in-place) ใช้พื้นที่ในหน่วยความจำหลักเพิ่มเป็นร้อยละเทียบกับขนาดข้อมูลจริงเพียงเล็กน้อย นอกจากนี้จะมีการใช้คู่สแต็กช่วยและขวาในการเปรียบเทียบและสลับ ในรูปแบบ Hoare's ซึ่งใช้แบนด์วิดท์ ของหน่วยความจำไม่มาก มีการนำเสนอการจัดการงาน (Scheduling) เพื่อให้การใช้งานแคชมีประสิทธิภาพมากขึ้น ถัดจากนั้นจะเป็นการอธิบายอัลกอริธึมที่มีการนำเสนอ ไม่ว่าจะเป็นการปรับขนาดบล็อก การแก้ไขปัญหาลำดับ (Skew pivot) ด้วยการปรับอัตราส่วนจำนวนเรดในการแบ่งข้อมูล การจัดการงานโดยใช้อัลกอริธึม Depth First Traversal การ

ปรับลดหรือเพิ่มจำนวนเรดระหว่างการแข่งขันข้อมูล ซึ่งสิ่งเหล่านี้จะช่วยแก้ไขปัญหการเข้าถึงหน่วยความจำ เพื่อให้การประมวลผลแบบขนานมีประสิทธิภาพสูงสุด

1.2 ความมุ่งหมายและวัตถุประสงค์

1.2.1 เพื่อศึกษาอัลกอริธึมการจัดเรียงข้อมูลแบบขนานที่มีการประมวลผลบนซีพียูหลายแกนโดยใช้สแต็ก

1.2.2 เพื่อพัฒนาอัลกอริธึมการจัดเรียงข้อมูลแบบขนานแบบบล็อกที่มีการประมวลผลบนซีพียูหลายแกนด้วยไลบรารีโอเพนเอ็มพี

1.2.3 เพื่อออกแบบอัลกอริธึมการจัดเรียงข้อมูลแบบขนานแบบบล็อกที่มีการประมวลผลบนซีพียูหลายแกน ให้เวลาที่ใช้ในการทำงานน้อยลง และมีความเสถียรมากขึ้น

1.3 ประโยชน์ที่คาดว่าจะได้รับ

1.3.1 เข้าใจหลักการออกแบบอัลกอริธึมการจัดเรียงข้อมูลแบบขนานที่มีการประมวลผลบนซีพียูหลายแกนโดยใช้สแต็ก

1.3.2 อัลกอริธึมใหม่สำหรับการจัดเรียงข้อมูลแบบขนานแบบบล็อกที่มีการประมวลผลบนซีพียูหลายแกนด้วยไลบรารีโอเพนเอ็มพี

1.3.3 อัลกอริธึมใหม่สำหรับการจัดเรียงข้อมูลแบบขนานแบบบล็อกที่มีการประมวลผลบนซีพียูหลายแกนที่ได้รับการออกแบบให้เวลาที่ใช้ในการทำงานน้อยลง และมีความเสถียรมากขึ้น

1.4 ขอบเขตของวิทยานิพนธ์

1.4.1 งานวิทยานิพนธ์นี้จะมีการพัฒนาอัลกอริธึมการจัดเรียงข้อมูลแบบขนานแบบบล็อก ที่ทำงานบนซีพียูหลายแกนด้วยไลบรารีโอเพนเอ็มพี บนระบบปฏิบัติการลินุกซ์เท่านั้น

1.4.2 งานวิทยานิพนธ์นี้มีการวัดประสิทธิภาพในการทำงานของอัลกอริธึมเฉพาะเวลาโดยวัดจากฟังก์ชัน `omp_get_wtime()` ที่ใช้ในการประมวลผลการจัดเรียงข้อมูล และความเสถียรของเวลาในการจัดเรียงข้อมูลเท่านั้น

1.4.3 ชุดข้อมูลที่ใช้ในการจัดเรียงที่มีการกระจายตัวแบบยูนิฟอร์ม (Uniform) และการวัดประสิทธิภาพของอัลกอริธึมด้วย Standard Template Library

1.4.4 การทดลองของการจัดเรียงข้อมูลใช้คอมไพเลอร์ GNU G++ โดยใช้แฟลก -O2 เพื่อออกแบบอัลกอริธึมให้มีประสิทธิภาพสูงสุด

1.5 ขั้นตอนการดำเนินงาน

ในงานวิทยานิพนธ์นี้จะศึกษาอัลกอริธึมการจัดเรียงข้อมูลแบบขนานที่มีการประมวลผลบนซีพียูหลายแกน โดยเริ่มศึกษาจากอัลกอริธึมการจัดเรียงข้อมูลแบบตามลำดับ การแบ่งข้อมูลชนิดต่างๆ เพื่อให้เข้าใจถึงหลักการทำงานของ การจัดเรียงข้อมูล จากนั้นจะมีการวิเคราะห์บริเวณที่สามารถพัฒนาอัลกอริธึมให้มีความทำงานแบบขนาน โดยจะแยกเป็นส่วนของการแบ่งข้อมูล ซึ่งจะมีการแบ่งข้อมูลแบบบล็อก และมีการเปรียบเทียบและสลับข้อมูล โดยใช้ MSPPartition ซึ่งภายในอัลกอริธึมนี้จะมีการประยุกต์โอเพนเอ็มพีซึ่งเป็นไลบรารีในการพัฒนาอัลกอริธึมให้มีความทำงานแบบขนานแบบเธรด (Thread-based) มาพัฒนาอัลกอริธึมนี้ โดยการทดลองของวิทยานิพนธ์นี้จะมีการเปรียบเทียบประสิทธิภาพการทำงานใช้บนซีพียูหลายแกน กับ BQSort MWSort ซึ่งเป็นอัลกอริธึมมาตรฐานในการจัดเรียงข้อมูลแบบขนานใน Standard Template Library

หลังจากที่มีการพัฒนาอัลกอริธึมการจัดเรียงข้อมูล MSPSort แล้ว ถัดจากนั้นจะเป็นการออกแบบอัลกอริธึมให้ทำงานมีประสิทธิภาพมากขึ้น โดยการปรับค่าพารามิเตอร์ต่างๆ ไม่ว่าจะเป็นการปรับขนาดบล็อก การแก้ไขปัญหาไฟวอทเอียง การปรับอัตราส่วนเธรดในการแบ่งข้อมูล การจัดการงานโดยใช้อัลกอริธึม Depth First Traversal การปรับจำนวนเธรดระหว่างการแบ่งข้อมูล และวัดประสิทธิภาพการทำงานแบบขนาน และพัฒนาให้อัลกอริธึมการจัดเรียงข้อมูลมีประสิทธิภาพในการทำงานสูงสุด

บทที่ 2

บททบทวนวรรณกรรม

ในบททบทวนวรรณกรรมจะมีการนำเสนอการจัดเรียงข้อมูลโดยใช้วิธีการแบ่งแยกและเอาชนะแบบขนาน และ STLSort โหมดการทำงานแบบตามลำดับและแบบขนาน

2.1 การจัดเรียงข้อมูลโดยใช้ขั้นตอนวิธีแบ่งแยกและเอาชนะแบบขนาน

ในการจัดเรียงข้อมูลแบบแบ่งแยกและเอาชนะมีหลักการคือแบ่งข้อมูลให้มีขนาดเล็กลงก่อน (Divide) เมื่อมีขนาดเล็กมากพอจากนั้นจัดเรียงอาเรย์นั้น (Conquer) ดังนั้นการจัดเรียงข้อมูลลักษณะนี้จะแบ่งช่วงออกเป็น 2 ช่วง คือ การแบ่งข้อมูล เป็นช่วงที่จะต้องมีการเปรียบเทียบข้อมูลและสลับข้อมูลแบบขนาน ซึ่งการแบ่งข้อมูลแบบขนานนั้นมักจะต้องมีการทำซิงโครไนเซชัน (Synchronization) และจะต้องมีการทำอัลกอริธึมคลีนอัป (Cleanup Algorithm) เพื่อให้ตำแหน่งของข้อมูลหลังจากการแบ่งข้อมูลแบบขนานอยู่ในตำแหน่งที่ถูกต้อง โดยการเปรียบเทียบข้อมูลนั้นจะมีการเปรียบเทียบข้อมูลในตำแหน่งที่อยู่ห่างกัน ซึ่งจะมีข้อจำกัดเกิดขึ้นที่เกิดจาก ทีแอลบี (Translation Look aside buffer: TLB) และเพจเทเบิล (Page Table) ซึ่งสิ่งเหล่านี้จะทำให้เกิดแคชมิส (Cache misses) ทั้งหมดนี้เป็นปัญหาที่เกิดจากข้อจำกัดของหน่วยความจำ (Memory bound problem) โดยในช่วงนี้อาจจะมีการจัดการงานเพื่อให้การแบ่งข้อมูลมีประสิทธิภาพมากยิ่งขึ้น ในช่วงถัดไปจะเป็นการจัดเรียงข้อมูล โดยจะเป็นการจัดเรียงข้อมูลแยกออกไป 1 เรด เมื่อจัดเรียงข้อมูลแล้ว ตำแหน่งข้อมูลเหล่านั้นจะอยู่ในตำแหน่งที่ถูกต้อง งานวิจัยที่เกี่ยวข้องกับการจัดเรียงข้อมูลโดยขั้นตอนวิธีการแบ่งแยกและเอาชนะแบบขนานมีหลายงาน ดังต่อไปนี้

ในปี 1990 Heidelberg et al. [3] ได้มีการนำเสนอผลลัพธ์ของการจัดเรียงข้อมูลด้วยอัลกอริธึมคิวคิวซอร์ตแบบขนาน โดยใช้วิธีการ Fetch-and-Add (F&A) งานนี้มีรูปแบบอิน-เพลส (in-place) และในการทำช่วงคลีนอัป (clean up) จะใช้พื้นที่บัฟเฟอร์ในการสลับข้อมูล ในช่วงถัดไปจะมีการใช้โครงสร้างข้อมูลคิวคิวโดยในคิวแรกจะเก็บอาเรย์ย่อยที่ยาวกว่าในการแบ่งข้อมูลแบบขนานซึ่งจะมีค่าตามจำนวนของหน่วยประมวลผล คิวที่สองจะเก็บอาเรย์ย่อยที่สั้นกว่าเพื่อที่จะแบ่งข้อมูลแบบตามลำดับ เมื่อข้อมูลถูกแบ่งแล้วจะนำมาจัดเรียงข้อมูล โดยความเร็วที่เพิ่มขึ้นจากการจำลอง (Simulation) ได้ถึง 400 เท่า เมื่อมีการจัดเรียงข้อมูลที่ 2^{20} ข้อมูล โดยใช้หน่วยประมวลผล 500 ตัว

ในปี 2003 Tsigas และ Zhang [4] ได้นำเสนอการจัดเรียงข้อมูลที่มีการแบ่งข้อมูลแบบบล็อก ที่เรียกว่า PQuickSort โดยขนาดบล็อกมีขนาดเล็กเท่ากับแคชระดับที่ 1 เท่านั้น ซึ่งแต่ละหน่วยประมวลผลจะประมวลผลคู่บล็อกด้านซ้ายและด้านขวาเพื่อเปรียบเทียบและสลับ เมื่อเปรียบเทียบสลับคู่บล็อกแล้วจะขยับมาทำบล็อกถัดไปทั้งด้านซ้ายและด้านขวา จนกระทั่งไม่เหลือ

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

บล็อกที่ยังไม่ได้เปรียบเทียบและสลับ บล็อกที่เปรียบเทียบและสลับไม่สำเร็จจะถูกย้ายเข้ามาใกล้กัน และแบ่งข้อมูลแบบตามลำดับจนสำเร็จ จากนั้นช่วงถัดไปจะมีการแบ่งอาเรย์ย่อยให้แต่ละหน่วยประมวลผล และสร้างสแต็กขึ้นมาเพื่อเก็บอาเรย์ย่อยเพื่อแบ่งข้อมูล และเรียงข้อมูลหลังจากที่แบ่งข้อมูลเรียบร้อยแล้ว โดยงานนี้ความเร็วที่เพิ่มขึ้นสูงถึง 11 เท่า บนเครื่อง SUN-T1 ที่มีหน่วยประมวลผล 32 ตัว

ในปี 2004 Süß และ Leopold [5] นำเสนอ อัลกอริธึมการจัดเรียงข้อมูลแบบขนานโดยใช้ Pthread และโอเพนเอ็มพี 2.0 โดยทดลองบนเครื่อง AMD Opteron 848 ซึ่งมี 4 แกน โดยสามารถทำความเร็วที่เพิ่มขึ้นได้ถึง 3.24 เท่า

ในปี 2007 Singler et al. [6,7] ได้พัฒนา Multi-Core Standard Template Library (MCSTL) ซึ่งสามารถทำงานได้บนภาษา C++ โดยหลักการของอัลกอริธึมการจัดเรียงข้อมูลแบบขนานมีรูปแบบใกล้เคียงกับงานของ Tsigas และ Zhang [4] โดยอัลกอริธึมนี้จะใช้ double-ended queue หรือ deque เป็นโครงสร้างข้อมูลในการจัดการงานในการจัดเรียงข้อมูลแบบขนาน ความเร็วที่เพิ่มขึ้นสามารถทำได้ถึง 21 เท่า เมื่อทดลองบนเครื่อง SUN-T1 ซึ่งมี 8 แกน 32 เธรด

ในปี 2008 Frias และ Petit [8] ได้ใช้วิธีการเฟซและเพิ่ม เช่นกัน แต่อย่างไรก็ตามในงานวิจัยนี้ได้มีการเพิ่มอัลกอริธึมคลีนอัปเดตใหม่ที่จะลดปริมาณการเปรียบเทียบและสลับ (C&S) โดยใช้โครงสร้างข้อมูลไบนารีทรีในการจัดการบล็อกที่ยังเปรียบเทียบไม่เสร็จซึ่งทำให้ความสมดุลของข้อมูลในทุกซีพียูมีปริมาณที่เท่ากัน ในปีเดียวกัน Traoré et al. [9] ได้พัฒนา DFWSort ซึ่งการจัดเรียงข้อมูลแบบ Introspective ให้มีการทำงานแบบขนาน มีการนำเอาเทคนิคการขโมยงาน เข้ามาใช้ในการจัดการงานการแบ่งข้อมูลแบบบล็อก และไม่จำเป็นต้องนำเอา deque เข้ามาใช้ ซึ่งความเร็วที่เพิ่มขึ้นสามารถทำได้ถึง 8.1 เท่าบน 16 หน่วยประมวลผล ในปี 2009 Ayguade et al. [10] นำเสนอ อัลกอริธึม MultiSort ซึ่งมีรูปแบบของอัลกอริธึมเป็นเมิร์จซอร์ต (Merge Sort) โดยจะมีการแบ่งอาเรย์ที่รับเข้ามาออกเป็นส่วนขนาดเท่ากัน จากนั้นใช้ควิกซอร์ตในการจัดเรียงข้อมูลเหล่านั้น จากนั้นจึงรวมข้อมูล (Merge) กลับเข้ามา โดยวิธีการนี้ใช้ omp task ใน โอเพนเอ็มพี 3.0 โดยความเร็วที่เพิ่มขึ้นสูงถึง 13.6 เท่าบนเครื่องที่มีหน่วยประมวลผล 32 แกน ในปีเดียวกัน Man et al. [11,12] ได้พัฒนา psort() ซึ่งเป็นอัลกอริธึมที่ถูกพัฒนาโดยรวมเอาควิกซอร์ตและเมิร์จซอร์ตเข้าด้วยกัน โดยความเร็วที่เพิ่มขึ้นถึง 11 เท่า บนเครื่อง Intel Xeon 7360 ซึ่งมีแกนประมวลผล 24 แกน

ในปี 2010 มีการนำเอาควิกซอร์ตมาทำงานบนหน่วยประมวลผลกราฟิกครั้งแรก [13] โดยงานนี้เป็นงานที่ไม่ใช้อิน-เพลส โดยประกอบไปด้วย 2 ส่วน ส่วนแรกจะแบ่งข้อมูลลงไปในโลคอลเมมโมรี ของจีพียู จากนั้นจะเปรียบเทียบและสลับกันในโลคอลเมมโมรีนั้น ผลลัพธ์ที่ได้จะถูกคัดลอกออกมา โดยในส่วนถัดไปจะแบ่งข้อมูลแบบรีเคอร์ซีฟโดยใช้สแต็กในการเก็บอาเรย์ย่อย และเมื่อขนาดเล็กพอจะนำอาเรย์เหล่านั้นไปจัดเรียงข้อมูล ในปี 2013 Mahafzah [14] ได้มีการพัฒนาอัลกอริธึมการจัดเรียงข้อมูล โดยในงานนี้จะมีลักษณะหลายไพวอต (pivot) โดยจะแบ่งข้อมูลเป็นส่วน โดยใช้พื้นที่เพิ่ม จากนั้นจัดเรียงข้อมูลที่ถูกแบ่งแล้วเหล่านั้นด้วย 8 เธรด โดยความเร็วที่เพิ่มขึ้นสูงถึง 3.8

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับใช้ในงานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ทางการค้า

ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

เท่าบนเครื่อง 2 แกน 4 เธรด หลังจากนั้น Ranokpanuwat และ Kittitornkun [15] ได้พัฒนา อัลกอริธึม Parallel Partition and Merge QuickSort (PPMQSort) โดยสามารถทำความเร็วที่ เพิ่มขึ้นได้ถึง 12.29 เท่า เมื่อเปรียบเทียบกับ qsort() บนเครื่อง Intel Xeon E5520 ซึ่งประกอบไปด้วย 8 แกน 16 เธรด ในปี 2016 ถัดมาในปี 2017 Axtmann et al. [16] ได้มีการพัฒนา IPS⁴O ซึ่ง อัลกอริธึมนี้มีรูปแบบเป็นบั๊กเก็ตซอร์ต (Bucket Sort) ที่ทำงานหลายเธรดบนอาเรย์ที่นำเข้ามา โดยแต่ละเธรดจะแบ่งข้อมูลลงในแต่ละบั๊กเก็ตจำนวน k ขึ้นอยู่กับจำนวนไพอท ของการแบ่งข้อมูลนั้น เมื่อแบ่งข้อมูลแล้วจะเป็นการรวมข้อมูลลงบนอาเรย์ โดยหากข้อมูลมีขนาดเล็กพอจะมีการจัดเรียงข้อมูล ความเร็วที่เพิ่มขึ้นสูงถึง 29 เท่าบนเครื่อง Intel Xeon E5-2683 รุ่นที่ 4 ในปี 2018 Rattanatanurak [17] พัฒนา Dual Parallel Partition Sort (DPPSort) โดยมีการเปรียบเทียบ ความเร็วที่เพิ่มขึ้นกับ qsort() และ STLSort ได้ความเร็วที่เพิ่มขึ้น 5.95 และ 4.70 เท่า ตามลำดับ เมื่อจัดเรียงข้อมูลบนเครื่อง Intel i7-3770 โดยอัลกอริธึมในการจัดเรียงข้อมูลแบบขนาน ได้แสดงไว้ในตารางที่ 2.1

ตารางที่ 2.1 การเปรียบเทียบการจัดเรียงข้อมูลแบบขนานในเชิงของวิธีการแบ่งข้อมูล คอขวด การทำรีเคอร์ซีฟ เวลา และไลบรารี ที่ใช้

อัลกอริธึม	การแบ่งข้อมูล	คอขวด	รีเคอร์ซีฟ	เวลา	ไลบรารี
PQuickSort [4]	ละเอียด ขนาดแคช L1	สลับข้อมูล มา ตรง กลางแบบ ตามลำดับ	มี	$O\left(\frac{N}{c} \log \frac{N}{c} + \frac{N}{c}\right)$	Pthread
BQSort [6,7]	ละเอียด ขนาด แคช L1	สลับข้อมูล มา ตรง กลาง	มี	$O\left(\frac{N}{c} \log N + c \log c\right)$	OpenMP
MWSort [6,7]	-	รวมข้อมูล แบบขนาน	มี	$O\left(\frac{N}{c} \log N + (c \log c) \left(\log \frac{N}{c}\right)\right)$	OpenMP
DFWSort [9]	ละเอียด ขนาดแคช L1	สลับข้อมูล มา ตรง กลาง	มี	$O\left(\frac{N}{c} + \log^3 N\right)$	OpenMP

ตารางที่ 2.1 การเปรียบเทียบการจัดเรียงข้อมูลแบบขนานในเชิงของวิธีการแบ่งข้อมูล คอขวด การทำรีเคอร์ซีฟ เวลา และไลบรารี ที่ใช้ (ต่อ)

อัลกอริธึม	การแบ่งข้อมูล	คอขวด	รีเคอร์ซีฟ	เวลา	ไลบรารี
psort [11,12]	หยาบ ขนาด N/c	รวมข้อมูลแบบตามลำดับจากนั้น qsort	ไม่มี	$O(\frac{N}{c} \log \frac{N}{c} + N)$	OpenMP
PPMQSort [15]	หยาบ ขนาด $N/2$	สลับข้อมูลแบบตามลำดับ	มี	$O(\frac{N}{c} \log \frac{N}{2c} + N)$	OpenMP
IPS ⁴ O [16]	ละเอียด แบ่งเป็นบล็อก	ทำบัคเก็ตแบบอิน-เพลส	มี	NA	OpenMP
DPPSort [17]	หยาบ ขนาด $N/2$	สลับข้อมูลแบบตามลำดับ	มี	$O(\frac{N}{c} \log \frac{N}{2c} + N)$	OpenMP

งานนี้เป็นการจัดเรียงข้อมูลโดยมีการแบ่งข้อมูลแบบบล็อกซึ่งมีการเปรียบเทียบและสลับ แต่ละครดจะมีคู่สแต็กย่อยๆของตนเอง เป็นสแต็กด้านซ้ายและด้านขวา และแบ่งข้อมูลขนานกันโดยไม่ต้องใช้วิธีการเพชและเพิ่ม โดยงานนี้จะมีซิงโครไนเซชันเมื่อสแต็กด้านซ้ายหรือด้านขวาวางลง โดยจะมีการทำรีเคอร์ซีฟทำฟังก์ชัน MSPPartition ซ้ำแบบขนาน และแบ่งข้อมูลแบบตามลำดับในขั้นตอนสุดท้าย ในช่วงถัดไปจะเป็นการจัดการงาน และมีการจัดเรียงข้อมูลเป็นครั้งสุดท้าย โดยที่ไม่มีการขโมยงานระหว่างซีพียู

2.2 STLSort โหมตการทำงานแบบตามลำดับและขนาน

Standard Template Library หรือ STL Sort เป็นฟังก์ชันในการจัดเรียงข้อมูลแบบขนาน โดยสามารถจัดเรียงข้อมูลชนิดต่างๆ โไลบรารีนี้สามารถเรียกใช้ด้วยภาษา C++ ในรูปแบบดังนี้

```
std::sort(RandomAccessIterator first, RandomAccessIterator last);
```

เมื่อ first และ last เป็นตัวชี้ตำแหน่งแรกและตำแหน่งสุดท้าย ตามลำดับ

ฟังก์ชันนี้ปกติแล้วจะสามารถทำงานแบบตามลำดับนั้นคือ เมื่อมีการเรียกใช้ฟังก์ชันซีพียูจะประมวลผลได้เพียง 1 เธรด ซึ่งทำให้ทรัพยากรในระบบไม่ได้ถูกใช้ให้มีประสิทธิภาพ ดังนั้นจึงมีการพัฒนา GNU libstdc++ parallel mode [18] ซึ่งเป็นไลบรารีที่มีการพัฒนาฟังก์ชัน การจัดเรียงข้อมูลแบบขนาน โดยมี 2 ฟังก์ชัน ที่มีประสิทธิภาพคือ Balanced QuickSort (BQSort) และ Multiway Merge Sort (MWSort) โดยฟังก์ชันทั้งสองนี้จะถูกนำมาเปรียบเทียบในการทดลองของงานวิจัยนี้ด้วย ฟังก์ชันนี้จะต้องเรียกใช้ผ่าน <parallel/algorithm> และเรียกใช้ในรูปแบบดังนี้

```
__gnu_parallel::sort(RandomAccessIterator first, RandomAccessIterator last);
```

2.2.1 Balanced QuickSort

อัลกอริธึม Balanced QuickSort หรือ BQSort เป็นอัลกอริธึมการจัดเรียงข้อมูลที่มีรูปแบบในการแบ่งข้อมูลเดียวกันกับงานของ Tsigas และ Zhang [4] ในส่วนของการแบ่งข้อมูลอัลกอริธึมนี้ จะมีการเปรียบเทียบและสลับข้อมูลระหว่างคู่บล็อกด้านซ้ายและด้านขวาแบบขนานจนกระทั่งฝั่งใด ฝั่งหนึ่ง ไม่มีข้อมูลหลงเหลืออยู่ ข้อมูลที่หลงเหลืออยู่อีกฝั่งหนึ่งจะถูกพุกกลับไปยัง double ended queue (deque) เพื่อที่จะนำข้อมูลที่หลงเหลืออยู่ชุดนั้นขึ้นมาทำใหม่ในรอบการเปรียบเทียบและสลับข้อมูลถัดไป ซึ่งหากมีหน่วยประมวลผลใดว่างจะมีการขโมยงานจากอีกเธรดหนึ่งมาทำ เพื่อให้ประสิทธิภาพการทำงานของหน่วยประมวลผลสูงที่สุด เมื่อทำจนครบทุกบล็อกแล้วบล็อกที่มีข้อมูลหลงเหลืออยู่จะถูกย้ายมายังส่วนกลางของอาร์เรย์ จากนั้นจะแบ่งข้อมูลใหม่ และจะมีการจัดเรียงข้อมูลตอนท้ายหลังจากที่มีการแบ่งข้อมูลเรียบร้อยแล้ว อัลกอริธึมนี้เป็นอัลกอริธึมแบบอิน-เพลส (in-place) ซึ่งมีการจัดความสมดุลของข้อมูลโดยใช้การขโมยงาน เวลาที่ใช้ในการทำงานของอัลกอริธึมนี้ ขึ้นอยู่กับการกระจายตัวของข้อมูล

2.2.2 MultiWay Merge Sort

อัลกอริธึม MultiWay Merge Sort หรือ MWSort เป็นอัลกอริธึมการจัดเรียงข้อมูลที่มีพื้นฐานมาจากเมิร์จซอร์ต โดยจะมีการแบ่งข้อมูลก่อนเท่าๆ กัน จากนั้นจัดเรียงข้อมูลอาเรย์ย่อยเหล่านั้นด้วย STLSort เนื่องจากการจัดเรียงข้อมูลเหล่านั้นทำได้ง่ายเนื่องจากมีขนาดเล็ก หลังจากนั้นเอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปเผยแพร่โดยไม่ได้รับอนุญาตจากเจ้าของเอกสาร หากมีผู้ใดที่นำเอกสารนี้ไปเผยแพร่โดยไม่ได้รับอนุญาตจากเจ้าของเอกสาร เจ้าของเอกสารขอสงวนสิทธิ์ในการดำเนินคดีตามกฎหมาย

MWSort จะมีการเรียกฟังก์ชัน Parallel multiway merge ซึ่งเป็นอัลกอริธึมในการรวมข้อมูลแบบขนาน อย่างไรก็ตามการจัดเรียงข้อมูลที่มีพื้นฐานมาจากเมิร์จซอร์ต จะต้องใช้พื้นที่อาเรย์เพิ่มขึ้นเท่ากับขนาดของข้อมูลตั้งต้น ดังนั้น MWSort จะต้องใช้พื้นที่อย่างน้อย 2 เท่าของขนาดอาเรย์ตั้งต้น แต่อย่างไรก็ตามเวลาที่ใช้ในการจัดเรียงข้อมูลมีความเสถียรมากกว่าอัลกอริธึมควิกซอร์ต

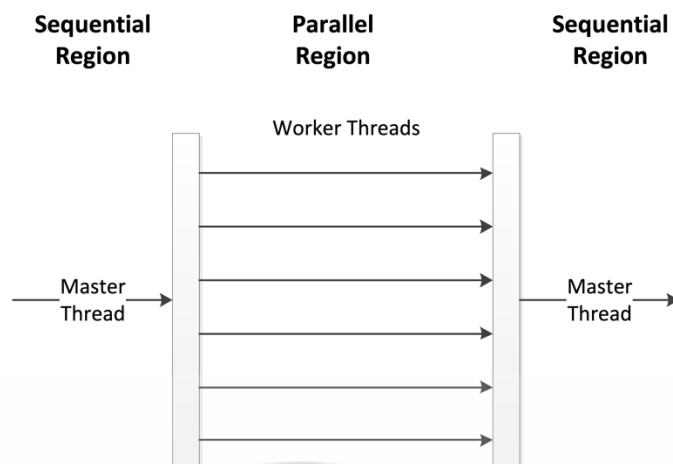
2.3 โอเพนเอ็มพี

ในการพัฒนาอัลกอริธึมให้มีการทำงานแบบขนานมากกว่าการประมวลผลแบบตามลำดับ อัลกอริธึมการจัดเรียงข้อมูล เป็นอัลกอริธึมหนึ่งที่เหมาะสำหรับการจัดเรียงข้อมูลทำงานแบบขนานจะสามารถดึงศักยภาพของระบบ ไม่ว่าจะเป็นหน่วยประมวลผล หน่วยความจำ และช่วยลดเวลาในการจัดเรียงข้อมูล ให้น้อยลง ซึ่งทำให้ประสิทธิภาพในการทำงานสูงมากยิ่งขึ้น

โอเพนเอ็มพี (OpenMP) [19] เป็นไลบรารี-เสริมโปรแกรมที่ใช้ในการพัฒนาให้มีการประมวลผลแบบขนานในระดับเธรด (Thread-based parallelism) บนหน่วยประมวลผลแบบแชร์หน่วยความจำ (Shared Memory MultiProcessor) ไลบรารีนี้ได้รับความนิยม และใช้งานอย่างแพร่หลายในการพัฒนาอัลกอริธึมให้ทำงานแบบขนานได้

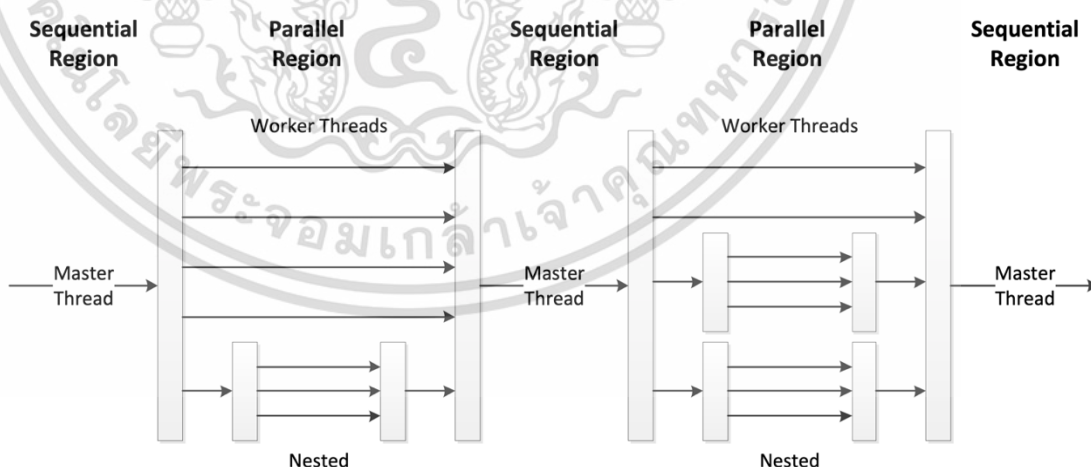
โอเพนเอ็มพี ออกแบบให้พัฒนาโปรแกรมได้บนภาษา C/C++ และภาษา Fortran โดยมีรัน-ไทม์ไลบรารี และตัวแปรสภาพแวดล้อม (Environment variables) ทำให้การพัฒนาโปรแกรมทำได้สะดวกและมีประสิทธิภาพ โดยสามารถเรียกใช้งานฟังก์ชันในไลบรารี และสามารถกำหนดตัวแปรสภาพแวดล้อมได้

การประมวลผลของโอเพนเอ็มพีมีรูปแบบเป็น ฟอว์ก/จอย โมเดล (Fork/Join model) โดยโมเดลนี้จะมีการทำงานแบบตามลำดับ ซึ่งจะทำงานด้วยมาสเตอร์เธรด (Master thread) จำนวน 1 เธรด เมื่อมีการเรียกใช้ `#pragma omp parallel` ซึ่งเป็นขอบเขตการทำงานแบบขนาน (Parallel region) จะมีการสร้างเธรด (Fork) เวิร์กเกอร์เธรด (Worker thread) ในจำนวนเธรดที่ถูกกำหนดไว้ก่อนหน้า ทำงานในขอบเขตที่มีการทำงานแบบขนาน เมื่อทำงานเสร็จ เธรดเหล่านี้จะถูกจอย (Join) กลับ และทำงานด้วยมาสเตอร์เธรด เพียง 1 เธรดเช่นเดิม รูปแบบของการทำงานแบบฟอว์ก/จอย โมเดล แสดงไว้ในภาพที่ 2.1 นอกจากนี้ไลบรารีโอเพนเอ็มพียังสามารถกำหนดขอบเขตการทำงานแบบขนานซ้อนกันได้ดังภาพที่ 2.2



ภาพที่ 2.1 ฟอว์ก/จอย โมเดลในโอเพนเอ็มพี

โครงสร้างการทำงานแบบขนาน (Parallel Construct) นั้นแบ่งออกเป็น 2 รูปแบบ ตามลักษณะการทำงานของประมวลผลแบบขนาน คือ โครงสร้างแบบขนานตามข้อมูล (Data Parallel Construct) และโครงสร้างแบบขนานตามงาน (Task Parallel Construct) โดยโครงสร้างแบบขนานตามข้อมูลจะเป็นการทำงานแบบขนานที่มีงานแบบเดียวกัน แต่ทำงานบนข้อมูลจำนวนมาก ซึ่งในงานการจัดเรียงข้อมูลจะเหมาะกับการเปรียบเทียบและสลับในรูปแบบบล็อก ในส่วนของโครงสร้างแบบขนานตามงาน จะเป็นการแยกเรดออกมาทำงานชนิดใดชนิดหนึ่ง ซึ่งอาจจะมีรูปแบบเดียวกัน หรือต่างรูปแบบกัน ซึ่งในงานการจัดเรียงข้อมูลจะเหมาะกับการทำรีเคอซีฟ เพื่อแบ่งข้อมูลในชั้นถัดไป หรือการจัดเรียงข้อมูล ภายหลังจากที่มีการแบ่งข้อมูลแล้ว



ภาพที่ 2.2 การฟอว์ก/จอย โมเดล แบบซ้อน ในโอเพนเอ็มพี

การใช้โอเพนเอ็มพีในการพัฒนาอัลกอริธึมให้ทำงานแบบขนานมีข้อดีคือ สามารถทำงานแบบขนานได้ง่ายและรวดเร็ว สามารถปรับจำนวนเรดให้สอดคล้องกับจำนวนแกนของซีพียูได้ สิ่งนี้ทำให้

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

โอเพนเอ็มพีเป็นหนึ่งในทางเลือกที่ดีในการนำเอามาประยุกต์ใช้ในการปรับอัลกอริธึมที่ทำงานแบบตามลำดับให้สามารถทำงานแบบขนาน เพื่อเพิ่มประสิทธิภาพของอัลกอริธึมให้สูงขึ้น



เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้าไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

บทที่ 3

การจัดเรียงแบบขนานโดยใช้หลายสแต็ก

ในบทนี้จะกล่าวถึงการจัดเรียงข้อมูลแบบขนานโดยใช้หลายสแต็ก (MultiStack Parallel Sort) โดยในบทนี้จะนำเสนออัลกอริธึม ซึ่งประกอบไปด้วย 2 ส่วน ได้แก่ Recursive MultiStack Parallel Phase และ Sorting Phase โดยในส่วนนี้จะมีการปรับปรุงการจัดกำหนดการของการแบ่งข้อมูลโดยจะมีการนำเสนอการจัดกำหนดการ (Scheduling) แบบ BF-DF และมีการเปรียบเทียบกับการจัดกำหนดการแบบอื่น

การจัดเรียงข้อมูลแบบขนานโดยใช้หลายสแต็ก นั้นจะมีการทำงานแบบขนาน ซึ่งอัลกอริธึมในการทำงานของการจัดเรียงข้อมูลแบบนี้ จะเริ่มต้นจากการหาตำแหน่งของไพวอท หรือตำแหน่ง p จากนั้นย้ายตำแหน่งไพวอทไปยังจุดกึ่งกลางของอาร์เรย์ A ถัดจากนั้น จะมีการเรียก Recursive MultiStack Parallel Phase ขึ้นมาเพื่อแบ่งข้อมูลด้วยไพวอทแบบขนาน หลังจากที่เราเรียกใช้ Recursive MultiStack Parallel Phase แล้ว จะได้ค่าตำแหน่งใหม่ของ p กลับออกมา (อัลกอริธึม 3.1, บรรทัดที่ 7) ถัดจากนั้นจะเป็นการจัดกำหนดการ (อัลกอริธึม 3.1, บรรทัดที่ 12 และ 15) หากอาร์เรย์ที่แบ่งได้มีความยาวน้อยกว่า n_{stl} อาร์เรย์ย่อยที่ได้จากถูกนำไปจัดเรียงข้อมูลด้วยเธรดอิสระ โดยใช้ task (อัลกอริธึม 1 บรรทัดที่ 20) โดยใช้ STLSort เมื่อ $n_{stl} = U_{stl} \times K_3 / \text{sizeof}(\text{Type})$ เมื่อ U_{stl} คือตัวแปรสัดส่วนการเรียงข้อมูลด้วย STLSort, K_3 เป็นขนาดของแคชระดับที่ 3 และ Type คือชนิดข้อมูล โดยที่จำนวนของซอฟต์แวร์เธรด จะมีค่าลดลงครึ่งหนึ่ง $\tau/2$ (อัลกอริธึม 3.1 บรรทัดที่ 9) จนกระทั่งมีค่าเท่ากับ $\tau = \tau_{max}/r$ เมื่อ τ_{max} คือค่าเธรดสูงสุดที่กำหนด และ r คือค่าสัดส่วนในการลดจำนวนเธรด

3.1 Recursive MultiStack Parallel Partition Phase

ในเฟสนี้ประกอบไปด้วย 2 ขั้นตอนย่อยคือ Parallel Stacked Blocks Partition Step และ Middle Blocks Partition Step

ใน Parallel Stacked Blocks Partition Step เริ่มต้นโดยการแบ่งข้อมูลอาร์เรย์ A ซึ่งประกอบไปด้วย $A[0], A[1], A[2], \dots, A[N-1]$ เมื่อ N คือจำนวนข้อมูลทั้งหมด ออกเป็น 2 ข้าง ด้านซ้ายและด้านขวา (อัลกอริธึม 3.2 บรรทัดที่ 8) จากนั้นแต่ละข้างจะแบ่งออกเป็นบล็อกข้อมูลขนาด b ตัว โดยที่ $b = B \times K_3 / \text{Type}$ เมื่อ B เป็นค่าพารามิเตอร์ขนาดของบล็อก จากด้านซ้ายสุดและขวาสุดของทั้งสองข้าง เข้ามายังกลางอาร์เรย์ A โดยแต่ละบล็อก จะถูกแบ่งแบบราวด์โรบิน (round robin) ไปยังอาร์เรย์ของสแต็กชื่อ L_s และชื่อ R_s โดยที่ L_s เป็นอาร์เรย์ของสแต็กด้านซ้าย และ

R_s เป็นอาร์เรย์ของสแต็กด้านขวา โดยจะมีการแบ่งข้อมูลจาก $L_s[0], L_s[1], \dots, L_s[\tau]$ ในด้านซ้าย และ

ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

$R_s[0], R_s[1], \dots, R_s[\tau]$ ในด้านขวา เช่นกัน โดยจะมีการpush ข้อมูลจากส่วนกลางของอาร์เรย์ไปยังด้านข้างแทน (อัลกอริธึม 3.2 บรรทัดที่ 4 และ 5) ดังนั้นจำนวนเรดและจำนวนบล็อก จะมีความสัมพันธ์กัน เพื่อให้ปริมาณงานแบ่งได้ใกล้เคียงกัน และทำให้การจัดเรียงข้อมูลแบบขนานสามารถทำได้มีประสิทธิภาพ

เมื่อข้อมูลได้pushลงสแต็กทั้งหมดเรียบร้อยแล้ว อัลกอริธึมจะมีการนำเอา OpenMP parallel for นำมาใช้ในการสร้างเรด จำนวน τ เรด (อัลกอริธึม 3.2 บรรทัดที่ 8) ซึ่งเท่ากับจำนวนขนาดของสแต็ก แต่ละข้าง โดยใช้ให้ตัวแปร i และ j ซึ่งเป็นตัวชี้ข้อมูลที่ใช้ในการเปรียบเทียบค่ากับค่าไพวอทส่วน l_b และ r_b ซึ่งเป็นขอบเขตของบล็อกด้านซ้าย และด้านขวาตามลำดับ เป็นตัวแปรประเภท private คือเป็นตัวแปรที่จะอยู่ภายในแต่ละเรด เรดอื่นไม่สามารถเข้าถึงได้ ถัดจากนั้นขอบเขตข้อมูลของแต่ละบล็อกจะถูกป๊อป (pop) ออกมาจากสแต็ก L_s และ R_s เพื่อทำการแบ่งข้อมูลกับ $A[p]$ หรือค่าไพวอทของการแบ่งข้อมูลครั้งนี้ หากข้อมูลด้านซ้ายมีค่ามากกว่า และข้อมูลด้านขวามีค่าน้อยกว่าค่าไพวอท จะมีการสลับข้อมูล จนกระทั่งตัวชี้ i หรือ j ของในแต่ละบล็อก มีค่ามากกว่า l_b และ r_b ตามลำดับ ซึ่งหากมีบล็อกที่ไม่สามารถแบ่งข้อมูลได้ครบ จะต้องถูกpushกลับไปยังสแต็ก (อัลกอริธึม 3.2 บรรทัดที่ 27 และ 30) และวนซ้ำจนกระทั่งสแต็กด้านซ้ายหรือขวาว่างครบทุกเรด

ถัดจากนั้น จะมีการหาค่าขอบเขตหลังจากการแบ่งข้อมูลของแต่ละเรด โดยจะหาค่าขอบที่ไม่ได้แบ่งข้อมูลด้านซ้ายสุด และด้านขวาสุด ดังนี้ $l_{min} = \min(L_s[t], \forall t)$ และ $r_{max} = \max(R_s[t], \forall t)$ โดยจะนำทุกเรดมาเปรียบเทียบกัน จากนั้นจะได้ขนาดของข้อมูลที่ยังไม่ได้ทำการแบ่งใหม่ โดยการนำ $r_{max} - l_{min}$ และนำไปเปรียบเทียบกับ u_{stl} (อัลกอริธึม 3.2 บรรทัดที่ 37) หากค่ายังมากกว่าจะมีการทำขั้นตอน Middle Blocks Partition Step โดยยังสามารถทำงานแบบขนานได้ตามจำนวน μ เรด ซึ่งจะมีการเรียก MSPPartition() (อัลกอริธึม 3.2 บรรทัดที่ 38) ซ้ำอีกครั้ง แต่หากค่า $r_{max} - l_{min}$ แล้วมีค่าน้อยกว่า u_{stl} จะมีการทำ Lomuto's Partition เพื่อหาค่าตำแหน่งของไพวอท (อัลกอริธึม 3.2 บรรทัดที่ 41) เนื่องจากการแบ่งข้อมูลแบบ Lomuto's มีการเข้าถึงหน่วยความจำน้อยกว่า Hoare's

3.2 Sorting Phase

ในส่วนก่อนหน้านี้เป็นส่วนที่มีการแบ่งข้อมูลออกเป็นส่วนย่อย วนซ้ำไปจนกระทั่งความยาวของข้อมูลของแต่ละส่วนย่อยน้อยกว่า u_{stl} จะมีการเปลี่ยนขั้นตอนการทำงานจากการแบ่งข้อมูลเป็นการจัดเรียงข้อมูลโดยใช้ STLSort โดยมีการแยกงานออกมา (อัลกอริธึม 3.1 บรรทัดที่ 20 และ 42) โดยไม่ต้องมีการทำซิงโครไนเซชัน ผ่าน OpenMP nowait

3.3 BF-DF Scheduling Algorithms

ใน Recursive MSPPartition Phase จะอาศัยการจัดงานของ OpenMP ตามปกติ อยู่ใน รูปแบบ BF (Breadth First) คือ จะมีการแบ่งข้อมูลไปพร้อมกันแบบขนาน เพื่อที่จะให้ทำงานได้อย่างมีประสิทธิภาพ แต่ปัญหาใหญ่ของการจัดงานแบบ BF คือ การเลือกงานขึ้นมาทำแบบสุ่ม ซึ่งขึ้นอยู่กับ ขนาดของข้อมูลที่ถูกแบ่งไว้แล้ว และการติดขัดจากบรานช์และหน่วยความจำ (branch/memory stall) ซึ่งสิ่งนี้จะทำให้เกิด เพจฟอลท์ (Page faults) และแคชมิส (cache misses) ปัญหาเหล่านี้ สามารถหลีกเลี่ยงได้ โดยจะต้องมีการพัฒนาอัลกอริธึมในการจัดลำดับงาน โดยประยุกต์ใช้ DF (Depth First) เข้าไปในฟังก์ชัน DFMSPSort ซึ่งทำให้เรดที่เกิดจากอัลกอริธึม BF ไม่มากเกินไป และกระบวนการแบ่งข้อมูลนี้จะทำงานอัลกอริธึม DF ต่อไป

ในอัลกอริธึมการจัดงานนี้จะเริ่มขึ้นโดยตรวจสอบความยาวของอาร์เรย์ย่อย ($j_R - i_L$) หากมีความยาวมากกว่า U_{df} (อัลกอริธึม 3.1 บรรทัดที่ 11) ฟังก์ชัน BFMSPSort จะถูกเรียกวนซ้ำ (อัลกอริธึม 3.1 บรรทัดที่ 12) โดยใช้ OpenMP task ซึ่งการวนซ้ำนี้จะถูกเรียกใช้จนกระทั่งความยาวของ $j_R - i_L$ มีค่าน้อยกว่า u_{df} โดยที่ $u_{df} = U_{df} \times K_{1/3}/\text{Type}$ เมื่อ U_{df} เป็นค่าตัวแปรในการเปลี่ยนการจัดงาน เมื่อมีค่าน้อยกว่าแล้ว จะมีการเรียกใช้ฟังก์ชัน DFMSPSort (อัลกอริธึม 3.1 บรรทัดที่ 15)

ในอัลกอริธึมจะมีสแต็ก P_s (อัลกอริธึม 3.1 บรรทัดที่ 32) ซึ่งจะเก็บขอบเขตของข้อมูลตามลำดับที่เกิดขึ้นจากการทำอัลกอริธึม DF ซึ่งนักพัฒนาสามารถประยุกต์ใช้การจัดงานแบบ DF โดยไม่ต้องเรียกใช้ฟังก์ชันจาก OpenMP ซึ่งการเก็บขอบเขตของข้อมูลโดยใช้สแต็กนี้ทำให้สามารถจัดเรียงการทำงานได้หลากหลายแบบ เช่น RAL LAL SPF และ LPF เพื่อเพิ่ม cache locality ของอัลกอริธึมการจัดเรียงข้อมูล

3.3.1 RAL vs LAL

ในส่วนนี้จะเป็นการนำเสนอการจัดการทำงานแบบ RAL และ LAL โดยเริ่มต้นจากข้อมูลส่วนแรกจะมีการพุ่งลงไปยังสแต็ก P_s (อัลกอริธึม 3.1 บรรทัดที่ 31) จากนั้นจะป๊อปออกมาเพื่อหาค่าขอบเขต i_L และ j_R จากนั้นส่งต่อไปยัง Recursive MSPPartition Phase (อัลกอริธึม 3.1 บรรทัดที่ 36) โดยเมื่อแบ่งข้อมูลแล้วจะมีการพุ่งข้อมูล ซึ่งจะมีอัลกอริธึมที่มีการพุ่งอาร์เรย์ย่อยด้านซ้ายลงไป ในสแต็กก่อน แล้วจึงพุ่งอาร์เรย์ย่อยด้านขวา เพื่อให้การแบ่งข้อมูลรอบถัดไปมีการป๊อปข้อมูลด้านขวาออกมาแบ่งข้อมูลก่อน การกำหนดลำดับการแบ่งข้อมูลเช่นนี้เรียกว่า Right Always: RAL การแบ่งข้อมูลเช่นนี้จะวนซ้ำไปจนกระทั่งส่วนของข้อมูลมีขนาดน้อยกว่า u_{stl} จากนั้น STLSort จะมีการทำงานอย่างอิสระโดยใช้ OpenMP nowait (อัลกอริธึม 3.1 บรรทัดที่ 43) และจะมีการเรียกป๊อปข้อมูลจากสแต็ก P_s จนกระทั่งสแต็กว่าง (อัลกอริธึม 3.1 บรรทัดที่ 32)

ในทางกลับกันอัลกอริธึมการจัดงานแบบ Left Always: LAL จะทำงานตรงกันข้ามกับ RAL โดยจะมีการพุ่งข้อมูลด้านขวาลงไปในสแต็ก P_s ก่อน จากนั้นจึงพุ่งข้อมูลด้านซ้ายลงไปในสแต็ก เมื่อมีการเรียกใช้ฟังก์ชันการแบ่งข้อมูล จะป๊อปข้อมูลด้านซ้ายขึ้นมาทำก่อนทำเช่นนี้วนซ้ำไปจนกระทั่งข้อมูลในสแต็ก P_s ว่าง

3.3.2 SPF vs LPF

โดยปกติแล้วการจัดงานแบบ RAL และ LAL จะมีการจัดการงานโดยขึ้นอยู่กับว่าข้อมูลอยู่ด้านซ้ายหรือด้านขวา ซึ่งเป็นการพิจารณาเฉพาะตำแหน่งเพียงเท่านั้น โดย RAL จะพुขข้อมูลด้านซ้ายก่อน และ LAL จะพुขข้อมูลด้านขวาก่อน แนวคิดหนึ่งที่สามารถจัดงานให้มีประสิทธิภาพมากขึ้นได้ โดยคำนึงถึงการจัดการหน่วยความจำแคช โดยจะมีการเลือกข้อมูลตามขนาด โดยมีขนาดสั้นกว่า (Shorter Partition First: SPF) และ ขนาดยาวกว่า (Longer Partition First: LPF) โดยจะมีการพुขข้อมูลลงสแต็กก่อน จากนั้นจะมีการเปรียบเทียบขนาดของข้อมูลเพื่อให้มีการเข้าถึงข้อมูลส่วนใหญ่ภายในแคช แต่อย่างไรก็ตาม การเลือกข้อมูลที่ยาวกว่าจะเป็นการเพิ่มเนื้องานเพื่อให้การประมวลผลแบบขนานมีได้มากขึ้นเช่นเดียวกัน

อัลกอริธึม 3.1 MSPSort

```

1 Function Main()
2     MSPSort(A, 0, N-1,  $\tau_{\max}$ )
3 End Function
4 Function MSPSort(A,  $i_L$ ,  $j_R$ ,  $\tau$ )
5      $p = \text{MO5}(A, i_L, j_R)$ 
6     If  $j_R - i_L > u_{stl}$  then
7          $p = \text{MSPPartition}(A, i_L, j_R, p, \tau)$ 
8         If  $\tau > \tau_{\max} / r$  then
9              $\tau = \tau / 2$ 
10        End
11        If  $j_R - i_L > u_{df}$  then
12            BFMSPSort(A,  $i_L$ ,  $j_R$ ,  $p$ ,  $\tau$ )
13        End
14        Else
15            DFMSPSort(A,  $i_L$ ,  $j_R$ ,  $p$ ,  $\tau$ )
16        End
17    End
18    Else
19        OpenMP Task
20        STLSort(A+ $i_L$ , A+ $j_R$ )
21    OpenMP nowait

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

อัลกอริธึม 3.1 MSPSort (ต่อ)

```

22     End
23 End Function

24 Function BFMSPSort(A,  $i_L, j_R, p, \tau$ )
25     OpenMP Task
26     MSPSort(A,  $i_L, p-1, \tau$ )
27     OpenMP Task
28     MSPSort(A,  $p+1, j_R, \tau$ )
29 End Function

30 Function DFMSPSort (A,  $i_L, j_R, p, \tau$ )
31      $P_s.push(i_L, j_R)$ 
32     While  $P_s$  not empty do
33          $(i_L, j_R) = P_s.pop()$ 
34         If  $j_R - i_L > u_{stl}$  then
35              $p = MO5(A, i_L, j_R)$ 
36              $p = MSPPartition(A, i_L, j_R, p, \tau)$ 
37              $P_s.push(i_L, p-1)$ 
38              $P_s.push(p+1, j_R)$ 
39         End
40     Else
41         OpenMP Task
42          $STLSort(A+i_L, A+j_R)$ 
43         OpenMP nowait
44     End
45 End
46 End Function

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

อัลกอริธึม 3.2 Parallel Stacked Blocks Partition

```

1 Function MSPPartition( $A, i_L, j_R, p, \tau$ )
2   halfB =  $(j_R - i_L) / (2b)$ 
3   For  $i = 0$  to halfB-1 do
4      $L_s[i \bmod \tau].push(i_L + i, i_L + i + b)$ 
5      $R_s[i \bmod \tau].push(j_R - i - b, j_R - i)$ 
6      $i = i + 1$ 
7   End
8   OpenMP parallel for private( $i, j, l_b, r_b$ )
9   For  $t = 0$  to  $\tau - 1$  do
10    While  $L_s[t]$  not empty AND  $R_s[t]$  not empty do
11       $(i, l_b) = L_s[t].pop()$ 
12       $(r_b, j) = R_s[t].pop()$ 
13      Do
14        While  $A[i] \leq A[p]$  AND  $i \leq l_b$  do
15           $i = i + 1$ 
16        End
17        While  $A[j] > A[p]$  AND  $j \geq r_b$  do
18           $j = j - 1$ 
19        End
20        If  $i \leq l_b$  AND  $j \geq r_b$  then
21           $R_s[t].push(r_b, j)$ 
22        End
23        Else if  $j < r_b$  then
24           $L_s[t].push(i, l_b)$ 
25        End
26      End
27    End
28  End
29   $l_{min} = \min(L_s[t], \forall t)$ 
30   $r_{max} = \max(R_s[t], \forall t)$ 
31   $\mu = (r_{max} - l_{min}) / u_{stl}$ 

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

อัลกอริธึม 3.2 Parallel Stacked Blocks Partition (ต่อ)

```

32   If  $r_{max} - l_{min} > u_{stl}$  then
33       return MSPPartition( $A, l_{min}, r_{max}, p, \mu$ )
34   End
35   Else
36       return LomutoPartition( $A, l_{min}, r_{max}, p$ )
37   End
38 End function

```



เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

3.4 การทดลอง ผลลัพธ์ และอภิปรายผลการทดลอง

หัวข้อนี้นำเสนอการทดลองอัลกอริธึมการจัดเรียงแบบขนานโดยใช้หลายสแต็กบนเครื่องระบบปฏิบัติการลินุกซ์ ซึ่งในหัวข้อนี้จะมีการนำเสนอการทดลอง ผลลัพธ์ที่ได้จากการทดลอง และอภิปรายผลการทดลอง

3.4.1 สภาพแวดล้อมในการทดลอง

ในการทดลอง MSPSort จะมีการทดลองบนเครื่องระบบปฏิบัติการลินุกซ์ 4 รูปแบบ ดังตาราง 3.1 โดยจะมีการติดตั้งระบบปฏิบัติการ Ubuntu 18.04 LTS และใช้คอมไพเลอร์ G++ 7.4 ซึ่งจะมีการเปรียบเทียบกันระหว่างซีพียูของ Intel และ AMD โดยจะมีการนำเสนอจำนวนแกนที่แสดงบนโปรแกรม System Monitor ของระบบปฏิบัติการ ขนาดของหน่วยความจำหลัก เทคโนโลยีที่ใช้ และการติดตั้ง นอกจากนี้จะใช้ซีพียูที่มีรูปแบบของหน่วยความจำแคช คล้ายคลึงกัน ซึ่งเป็นตัวคูณของ 8MB นอกจากนี้ภายในตารางมีการระบุรูปแบบ NUMA ซึ่งย่อมาจาก non-uniform memory access โดยเครื่อง R7-1700 จะประกอบไปด้วยตัวควบคุมหน่วยความจำหลัก 2 ตัวในแต่ละส่วนและเชื่อมต่อกันด้วย Infinity Fabric ซึ่งส่งผลให้มีรูปแบบการทำงานของหน่วยความจำแบบ non-uniform

ตารางที่ 3.1 เครื่องที่ใช้ในการทดลอง

ซีรี่ส์ รุ่น	Core i7 i7-2600	Xeon X5670	Ryzen R7-1700	ThreadRipper R9-2920
สัญญาณนาฬิกา (GHZ)	3.40	2.93	3.00	3.50
แกน	8	24	16	24
ซ็อกเก็ต	1	2	1	1
หน่วยความจำ หลัก	32 GB	24 GB	32 GB	64 GB
การติดตั้ง หน่วยความจำ หลัก	4 x 8GB	12 x 2GB	4 x 8GB	8 x 8GB
เทคโนโลยี	DDR3	DDR3	DDR4	DDR4
NUMA	ไม่ใช่	ใช่	มีลักษณะ เดียวกัน	ใช่
ช่องสัญญาณ	2ch	4ch	2ch	4ch
L1 I-Cache	4x32KB 8W	2x6x32KB 4W	8x64 4W	12x64KB 4W
L1 D-Cache	4x32KB 8W	2x6x32KB 8W	8x32KB 8W	12x32KB 8W
L2 Cache	4x256KB 8W	2x6x256KB 8W	8x512KB 8W	12x512KB 8W
L3 Cache	8MB 16W	2x12MB 16W	2x8MB 16W	4x8MB 16W

ในการทดลองนี้จะมีตัวแปรที่ใช้ดังตารางที่ 3.2 ในการทดลองนี้จะมีการทดลองจัดเรียงข้อมูล Unsigned 32-bit integer (Uint32) Unsigned 64-bit integer (Uint64) และ 64-bit double precision floating point (Double) โดยในการทดลองจะมีการสุ่มข้อมูลแบบ uniform ในการคอมไพล์ ใช้ข้อปติไมซ์ -O2 ทุกการทดลอง ขนาดข้อมูลที่ใช้ตั้งแต่ 200 ล้าน ถึง 2,000 ล้าน ทั้งนี้ขึ้นอยู่กับข้อจำกัดของหน่วยความจำหลักของแต่ละเครื่อง และมีการทดลองการจัดงานแบบ RAL LAL LPF และ SPF

ตัวแปรที่ใช้ในการทดสอบมีขนาดบล็อก B ขนาดของการจัดเรียงข้อมูลด้วย STLSort U_{stl} ขนาดในการเปลี่ยนวิธีการจัดงาน U_{df} โดยค่าเหล่านี้ จะเป็นจำนวนเท่าของ K_3 ซึ่งมีค่า 8MB โดย B จะมีค่าตั้งแต่ 10^{-4} , 10^{-3} , 10^{-2} , 10^{-1} , 1 ค่า U_{stl} มีค่าตั้งแต่ 0.5, 1, 2, 4, 8 ค่าของ U_{df} มีค่าตั้งแต่ 0.5, 1, 2, 4, 8, 16 ค่า m เป็นค่าของตัวคูณเรด โดยจะมีค่าตั้งแต่ 1, 2, 4 เพื่อให้มีการสร้างเรดใน

ช่วงแรกของการแบ่งข้อมูลมีค่ามากเพื่อให้ได้ $\tau_{\max} = c \times m$ และเมื่อสร้างเธรดในช่วงแรกแล้ว หลังจากนั้นจะต้องมีการลดจำนวนเธรดลง โดยใช้ ตัวลดเธรด r ซึ่งมีค่า $c, c/2, c/3, c/4$

ตารางที่ 3.2 ตัวแปรที่ใช้ในการทดลอง MSPSort

ตัวแปร	ค่าตัวแปร
อัลกอริธึม	MSPSort, BQSort, MWSort
ชนิดตัวแปร	Uint32, Uint64, Double
การกระจายข้อมูล	สุ่มแบบ Uniform
การ optimize ของ GCC	O2
ขนาดข้อมูล ($N \times 10^6$)	200, 500, 1000, 2000
การจัดงาน	RAL, LAL, LPF, SPF
ขนาดแคช $L3 \ K_{L3}$	8MB
ขนาดบล็อก $B \ (x \ K_{L3})$	$10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1$
$U_{stl} \ (x \ K_{L3})$	0.5, 1, 2, 4, 8
$U_{df} \ (x \ K_{L3})$	0.5, 1, 2, 4, 8, 16
ตัวคูณเธรด m	1, 2, 4
ตัวลดเธรด r	$c, c/2, c/3, c/4$

3.4.2 การวัดประสิทธิภาพ

ในงานนี้จะมีการทดลองเพื่อหาผลลัพธ์จะมีค่าที่ใช้ในการวัดประสิทธิภาพของอัลกอริธึม ซึ่งประกอบไปด้วย เวลาในการจัดเรียงข้อมูล ($\bar{T}$) เวลาในการจัดเรียงข้อมูลต่อข้อมูล 100 ล้านตัว ($\bar{T}_{100M}$) ค่าเบี่ยงเบนมาตรฐานของเวลาในการจัดเรียงข้อมูล (σ_T) และค่าเบี่ยงเบนมาตรฐานของเวลาในการจัดเรียงข้อมูลต่อข้อมูล 100 ล้านตัว (σ_{100M}) สถิติของเวลาในการจัดเรียงข้อมูล โดยรายละเอียดของค่าที่ใช้ในการวัดประสิทธิภาพมีดังต่อไปนี้

3.4.2.1 ค่าเฉลี่ยของเวลาในการจัดเรียงข้อมูล ($\bar{T}$) และเวลาในการจัดเรียงข้อมูลต่อข้อมูล 100 ล้านตัว ($\bar{T}_{100M}$)

การหาค่าเฉลี่ยของเวลาในการทดลองการจัดเรียงข้อมูลหลายๆ รอบ ซึ่งเป็นค่าที่สามารถวัดประสิทธิภาพได้โดยหาค่าเฉลี่ยของเวลามีค่าน้อย นั่นคืออัลกอริธึมการจัดเรียงข้อมูลนั้นมีประสิทธิภาพมากกว่า นอกจากนี้ยังมีค่าเฉลี่ยของเวลาในการจัดเรียงข้อมูลต่อข้อมูล 100 ล้านตัว ซึ่งทำให้ง่ายในการเปรียบเทียบประสิทธิภาพการทำงานของแต่ละการทดลอง ซึ่งสิ่งนี้จะมีการเปรียบเทียบกันระหว่างเครื่องที่ใช้ในการทดลองอีกด้วย

3.4.2.2 ค่าเบี่ยงเบนมาตรฐานของเวลา (σ_T) และค่าเบี่ยงเบนมาตรฐานของเวลาต่อข้อมูล 100 ล้านตัว (σ_{100M})

ค่านี้เป็นค่าแสดงความเสถียรของอัลกอริธึมเนื่องจากข้อมูลที่ใช้ทดลองเป็นการสุ่มขึ้น ดังนั้นจึงสามารถใช้ในการเปรียบเทียบเสถียรภาพของอัลกอริธึม ซึ่งเป็นสิ่งสำคัญในการจัดเรียงข้อมูล นอกจากนั้นยังนำเอาค่าเบี่ยงเบนมาตรฐานของเวลาต่อข้อมูล 100 ล้านตัว เพื่อใช้ในการปรับแต่งค่าตัวแปร เช่น B และ U_{stl}

3.4.2.3 สถิติของเวลาในการจัดเรียงข้อมูล

โดยปกติแล้วในทางคณิตศาสตร์การจับเวลาในการทำงานอาจจะใช้การหา ควอไทล์ ที่ 1 ควอไทล์ที่ 2 ควอไทล์ที่ 3 หรือ T_{Q1} , T_{Q2} และ T_{Q3} ตามลำดับ ซึ่งสามารถหาระยะอินเตอร์ควอไทล์ได้ดังนี้ $T_{IQR} = T_{Q3} - T_{Q1}$ โดยการทดลองเพื่อความวิเคราะห้ที่แม่นยำ จะมีการจับเวลาอย่างน้อย 1,000 รอบ

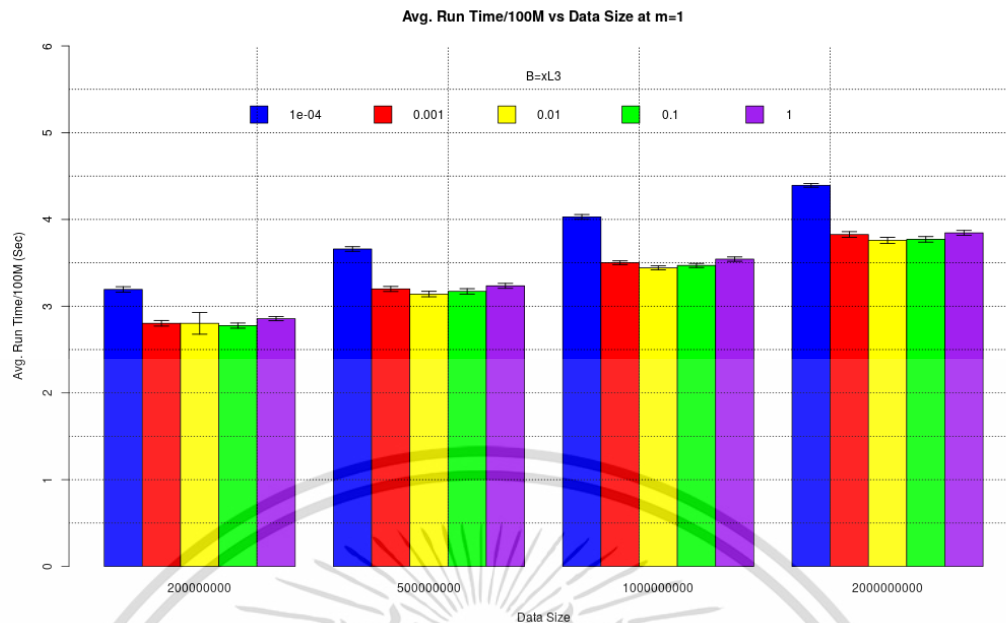
3.5 ผลการทดลองและการอภิปรายผลการทดลอง

3.5.1 การทดลอง MSPPartition ในการแบ่งข้อมูลครั้งแรก

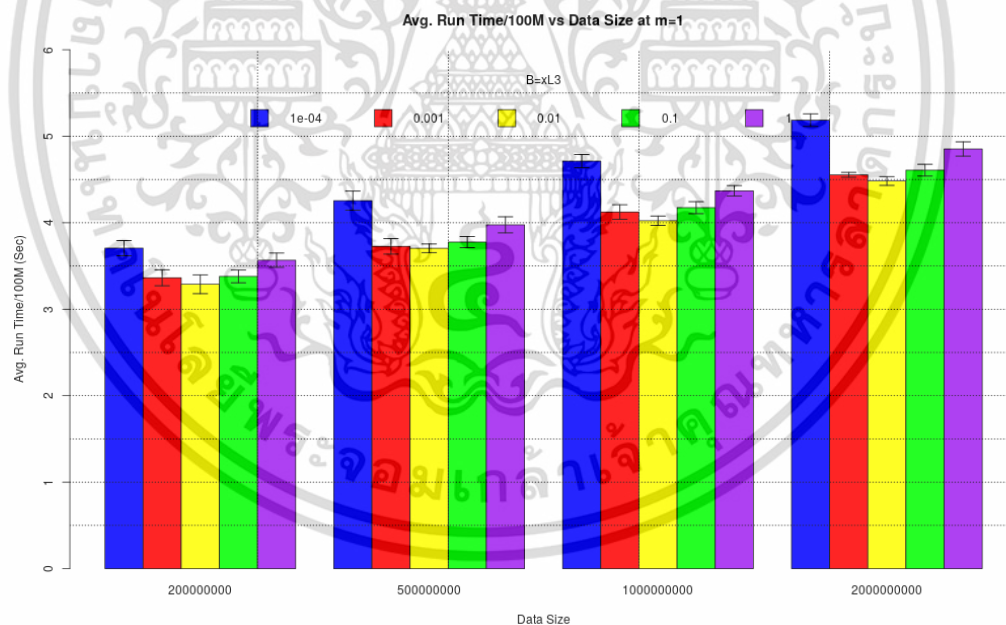
หลักการของการแบ่งข้อมูลจะต้องมีการแบ่งรอบแล้ว และลดขนาดอาเรย์ที่แบ่งได้ลงในรอบถัดไป โดยจะต้องมีการปรับค่าบล็อก B การแบ่งข้อมูลจะมีการปรับขนาดข้อมูลดังตารางที่ 3.2 โดยการทดลองจะหาความสัมพันธ์ระหว่าง B กับ การแบ่งข้อมูล (อัลกอริทึม 3.2 บรรทัดที่ 1) ซึ่งการทดลองนี้ข้อมูลบล็อกทางซ้ายและขวาจะสลับกันซึ่งจะทำให้เกิดบรานช์พรีดิกชัน (Branch prediction) ซึ่งเกิดจากการเปรียบเทียบ โดยการทดลองนี้ MSPPartition จะทำงาน 1 รอบ โดยไม่มีการเรียกซ้ำ ในการทดลองนี้จะกำหนดค่า B ให้อยู่ในช่วง $B = \{10^{-4}, 0.001, 0.01, 0.1, 1\} \times K_{L3}$ โดยค่าสูงสุดของจำนวนเรดคือ $T_{max} = c \times 1$ และไม่ใช้คีย์เวิร์ด OpenMP nested (omp_set_nested(0)) ซึ่งทำให้มองเห็นประสิทธิภาพของอัลกอริทึมได้อย่างแท้จริง

ผลลัพธ์ที่ได้ของ $\bar{T}_{100M}$ และ σ_{100M} มีหน่วยเป็นวินาที แสดงในภาพที่ 3.1-3.4 ในขนาดอาเรย์ N ที่แตกต่างกัน จำนวน 100 ครั้ง ทุกเครื่องแสดงให้เห็นผลลัพธ์ของ $\bar{T}_{100M}$ เปรียบเทียบกับ B ในลักษณะแบบเดียวกัน ซึ่งจะสังเกตเห็นว่าเมื่อขนาดของข้อมูลมีขนาดใหญ่ จะทำให้เกิด $\bar{T}_{100M}$ ที่สูงเช่นกัน สิ่งนี้เกิดจากการเข้าถึงแคช ซึ่งค่าแคชโลคอลิตี (Cache locality) เข้าถึงข้อมูลจากข้อมูลทั้ง 2 ฝั่งของอาเรย์ ซึ่งมีระยะห่างกันมาก ค่า $\bar{T}_{100M}$ จะได้ค่า B อยู่ระหว่าง 0.001 ถึง $0.1 \times K_{L3}$ ซึ่งอยู่ระหว่างขนาดของแคช L1 และ L2 จึงทำให้เลือกใช้ $B = 0.01 \times K_{L3}$ มาเป็นค่าที่ใช้ในการทดลอง

ในการทดลองนั้นจะมีการนำผลการทดลองมาสร้างเป็นกราฟ โดยใช้แกน Y เดียวกัน โดยตั้งค่า $m=1$ ซึ่งในแต่ละเครื่องจะได้ค่าจำนวนเรด c แตกต่างกัน กล่าวคือ i7-2600 จะมีค่า $\bar{T}_{100M}$ น้อยเมื่อเปรียบเทียบกับ X5670 เมื่อมีขนาดข้อมูล N เท่ากัน แม้ว่าจำนวนแกนมีค่าน้อยกว่า เช่นเดียวกันกับ R7-1700 จะมีค่า $\bar{T}_{100M}$ ที่เร็วกว่า R9-2920 เนื่องจากค่าสัญญาณนาฬิกาของซีพียู มีค่าน้อยกว่าและจำนวนแกนน้อยกว่า สิ่งนี้เนื่องจากซีพียู มีสถาปัตยกรรม Non-Uniform Memory Access บนเครื่อง X5670 และ R9-2920 ดังตารางที่ 3.1

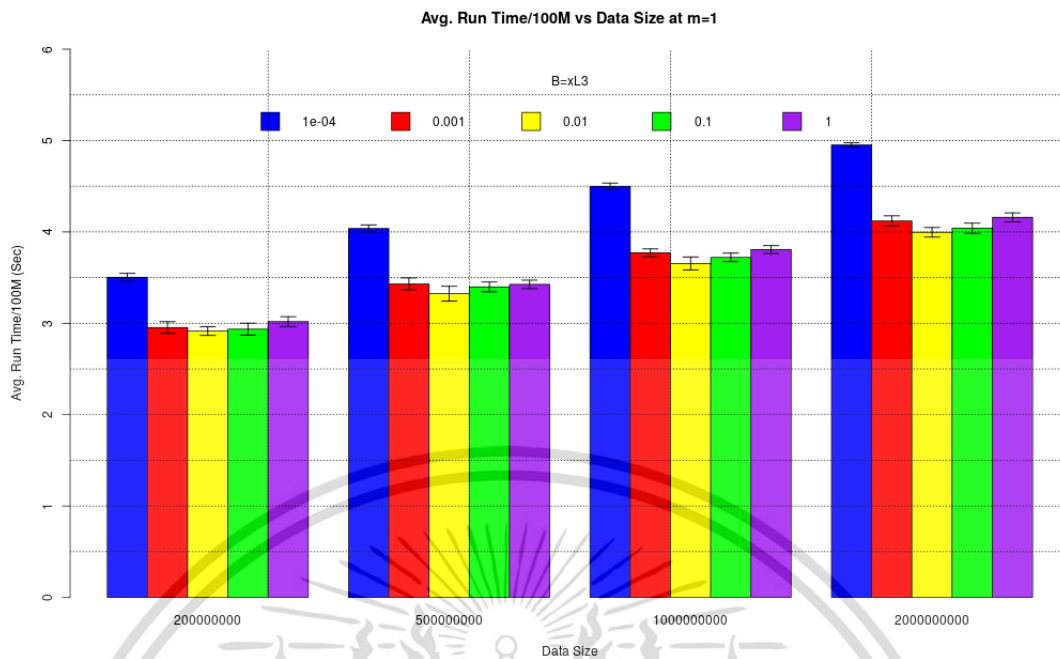


ภาพที่ 3.1 $\bar{T}_{100M}$ (กราฟแท่ง) และ $\pm\sigma_{100M}$ (กราฟความผิดพลาด) ของการแบ่งข้อมูลรอบแรก ที่ $B=\{10^{-4}, 0.001, 0.01, 0.1, 1\} \times K_{L3}$ ที่ $m=1$ บนเครื่อง i7-2600 เมื่อข้อมูลเป็น Uint32

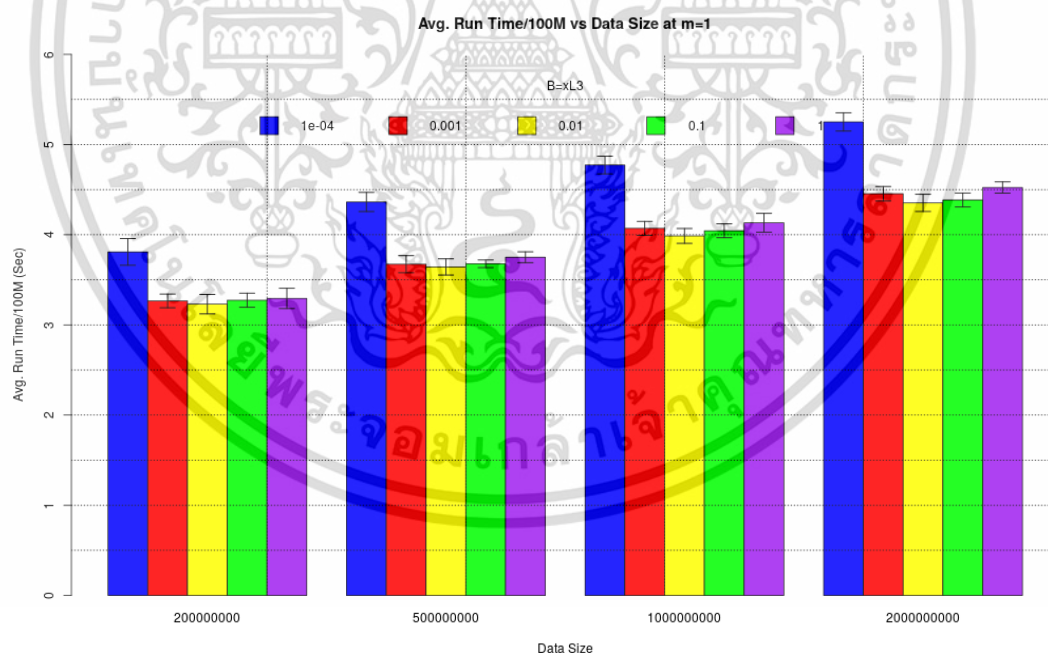


ภาพที่ 3.2 $\bar{T}_{100M}$ (กราฟแท่ง) และ $\pm\sigma_{100M}$ (กราฟความผิดพลาด) ของการแบ่งข้อมูลรอบแรก ที่ $B=\{10^{-4}, 0.001, 0.01, 0.1, 1\} \times K_{L3}$ ที่ $m=1$ บนเครื่อง X5670 เมื่อข้อมูลเป็น Uint32

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้



ภาพที่ 3.3 $\bar{T}_{100M}$ (กราฟแท่ง) และ $\pm\sigma_{100M}$ (กราฟความผิดพลาด) ของการแบ่งข้อมูลรอบแรก ที่ $B=\{10^{-4}, 0.001, 0.01, 0.1, 1\} \times K_3$ ที่ $m=1$ บนเครื่อง R7-1700 เมื่อข้อมูลเป็น Uint32



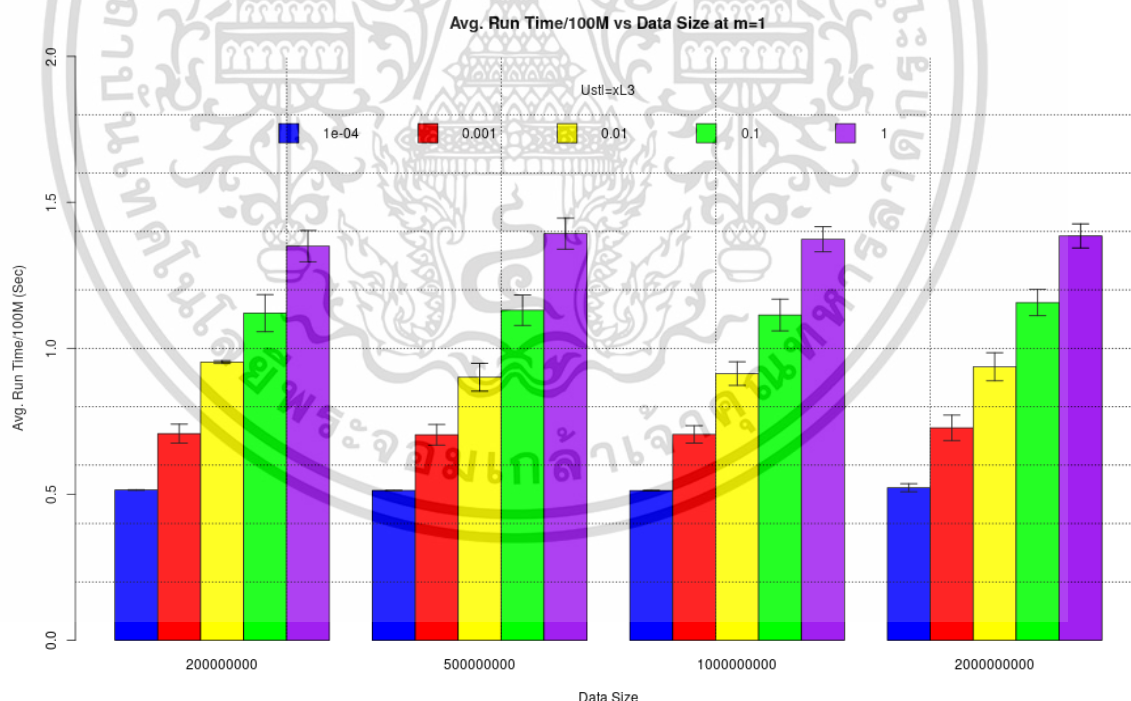
ภาพที่ 3.4 $\bar{T}_{100M}$ (กราฟแท่ง) และ $\pm\sigma_{100M}$ (กราฟความผิดพลาด) ของการแบ่งข้อมูลรอบแรก ที่ $B=\{10^{-4}, 0.001, 0.01, 0.1, 1\} \times K_3$ ที่ $m=1$ บนเครื่อง R9-2920 เมื่อข้อมูลเป็น Uint32

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

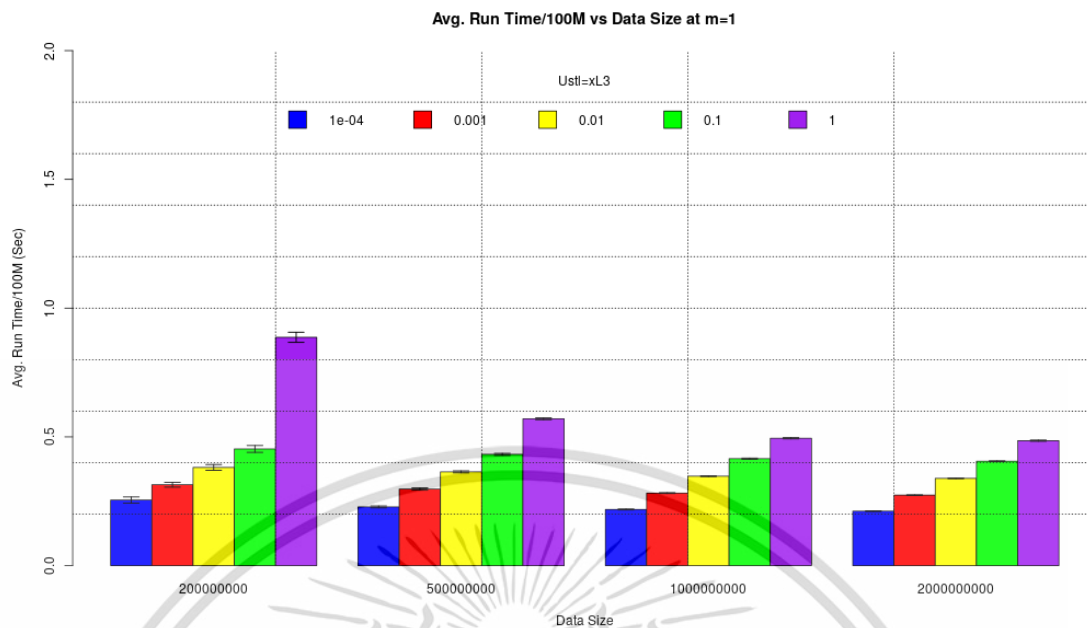
3.5.2 การทดลองจัดเรียงข้อมูลแบบอิสระกันของแต่ละบล็อก

ในการทดลองเพื่อหาค่า U_{stl} ที่เหมาะสม ในการหาค่าเวลาที่ใช้ในการจัดเรียงข้อมูลนั้น ข้อมูลขนาด N นั้นจะแบ่งออกเป็นส่วนย่อย ๆ ของ U_{stl} และส่งต่อให้แต่ละเธรด เพื่อจัดเรียงข้อมูลแบบขนาน โดยข้อมูลที่ถูกแบ่งแล้วจะได้รับการจัดเรียงโดย STLSort อย่างอิสระ โดยใช้ $c \times 1$ เธรด และค่า $m=1$ โดยจะปิด OpenMP nested เอาไว้ สิ่งนี้จะเป็นการทดสอบประสิทธิภาพของการจัดเรียงข้อมูลแบบ Divide and Conquer เนื่องจากเป็นการตัดการทำข้อมูลโดยไม่สนใจการแบ่งข้อมูลก่อนหน้า ซึ่งจะมีการสุ่มข้อมูลลงบนอาร์เรย์ขนาด N และตัดให้มีขนาดเท่ากับ โดยทดสอบ $U_{stl} = \{0.0001, 0.001, 0.01, 1\} \times K_{L3}$

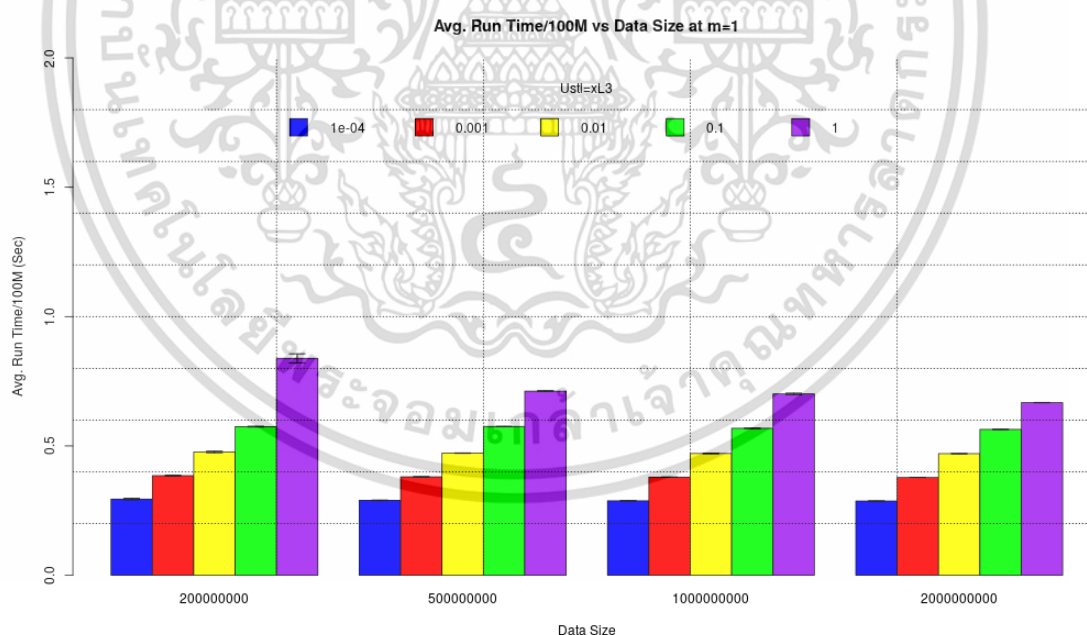
การทดลองนี้จะมีการทำซ้ำ 100 รอบ โดยจะมีการนำไปวาดกราฟ $\bar{T}_{100M}$ (กราฟแท่ง) และ σ_{100M} (กราฟความผิดพลาด) ดังภาพที่ 3.5-3.8 ซึ่งผลลัพธ์จากการทดลองจะเห็นว่าที่ขนาด N เท่ากัน เมื่อลดค่า U_{stl} ลง จะทำให้ $\bar{T}_{100M}$ มีค่าลดลง นั่นคือ U_{stl} ยิ่งมีค่าน้อย เป็นสิ่งที่ดีกว่า ในกรณีที่มีการจัดเรียงข้อมูลอิสระในแต่ละส่วน ซึ่งจะสอดคล้องกับการจัดเรียงข้อมูลในช่วงท้ายของอัลกอริธึมการจัดเรียงข้อมูลแบบขนาน



ภาพที่ 3.5 $\bar{T}_{100M}$ (กราฟแท่ง) และ σ_{100M} (กราฟความผิดพลาด) ของจัดเรียงข้อมูลแบบอิสระจากกัน ที่ $B=\{10^{-4}, 0.001, 0.01, 0.1, 1\} \times K_{L3}$ ที่ $m=1$ บนเครื่อง i7-2600 เมื่อข้อมูลเป็น Uint32

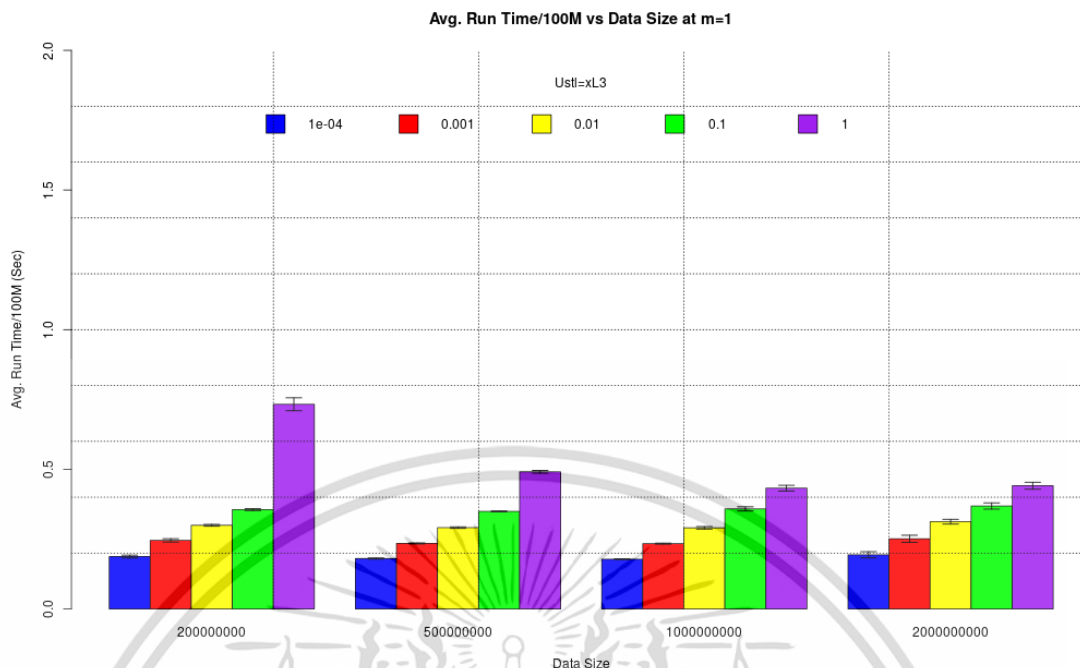


ภาพที่ 3.6 $\bar{T}_{100M}$ (กราฟแท่ง) และ $\pm\sigma_{100M}$ (กราฟความผิดพลาด) ของจัดเรียงข้อมูลแบบอิสระ จากกัน ที่ $B=\{10^{-4}, 0.001, 0.01, 0.1, 1\} \times K_3$ ที่ $m=1$ บนเครื่อง X5670 เมื่อข้อมูลเป็น Uint32



ภาพที่ 3.7 $\bar{T}_{100M}$ (กราฟแท่ง) และ $\pm\sigma_{100M}$ (กราฟความผิดพลาด) ของจัดเรียงข้อมูลแบบอิสระ จากกัน ที่ $B=\{10^{-4}, 0.001, 0.01, 0.1, 1\} \times K_3$ ที่ $m=1$ บนเครื่อง R7-1700 เมื่อข้อมูลเป็น Uint32

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้



ภาพที่ 3.8 $\bar{T}_{100M}$ (กราฟแท่ง) และ $\pm\sigma_{100M}$ (กราฟความผิดพลาด) ของจัดเรียงข้อมูลแบบอิสระ จากกัน ที่ $B=\{10^{-4}, 0.001, 0.01, 0.1, 1\} \times K_3$ ที่ $m=1$ บนเครื่อง R9-2920 เมื่อข้อมูลเป็น Uint32

3.5.3 MSPSort โดยจัดงานแบบ BF

การทดลองก่อนหน้านี้เป็นการทดลองที่เป็นการแยกส่วนประกอบของการจัดเรียงข้อมูล โดยการทดลองนี้จะเป็นการเอาส่วนประกอบการแบ่งข้อมูล และการจัดเรียงข้อมูลมาอยู่ด้วยกัน โดยการทดลองนี้จะมีการเปิด OpenMP nested และมีการเรียกใช้ฟังก์ชันวนซ้ำของอัลกอริธึม โดยการจัดงานแบบ BF นั้นจะแสดงในอัลกอริธึม 3.1 บรรทัดที่ 12 ในการทดลองนี้จะเป็นการปรับแต่งค่าพารามิเตอร์ที่มีจำนวนมาก ทำให้เกิดรูปแบบของพารามิเตอร์ที่ใช้ในการทดลอง โดยจะมีการเลือกค่าคู่ของ m และ r มาใช้ที่มีประสิทธิภาพของแต่ละเครื่อง โดยค่า $\bar{T}$ จะถูกทดลองโดยใช้การจัดงานแบบ BF ทุกขนาดข้อมูล N ที่ $B=0.01$, $U_{stl} = 0.5, 1, 2, 4$ โดย คู่ของ m และ r จะถูกเลือกมา โดยจะเลือกคู่ที่มีเวลาน้อย ของทุกขนาดข้อมูล ดังตารางที่ 3.3 โดยที่ คู่ของ m และ r ที่ดีที่สุดจะอยู่ด้านบนของตาราง ซึ่งสิ่งนี้จะถูกนำมาใช้ในการทดลองต่อในการทดลองถัดไป

ตารางที่ 3.3 คู่ m และ r 3 อันดับแรก ในการจัดงานแบบ BF ในทุกขนาดข้อมูล ที่ $B=0.01$ $U_{stl} = 0.5, 1, 2, 4$ โดยทดสอบ 20 รอบ

ระบบ	i7-2600	X5670	R7-1700	R9-2900
Uint32	(2, 8)	(1, 6)	(2, 16)	(2, 12)
	(2, 4)	(2, 12)	(2, 8)	(1, 8)
	(1, 4)	(1, 8)	(1, 16)	(1, 6)
Uint64	(2, 8)	(1, 6)	(2, 16)	(1, 8)
	(1, 4)	(2, 12)	(1, 8)	(1, 6)
	(2, 4)	(1, 8)	(2, 8)	(2, 12)
Double	(2, 8)	(1, 6)	(2, 16)	(1, 8)
	(1, 4)	(2, 12)	(1, 8)	(1, 6)
	(2, 4)	(1, 8)	(2, 8)	(2, 12)

3.5.4 MSPSort โดยจัดงานแบบ BF-DF

การทดลองนี้จะเป็นการนำอัลกอริธึมที่ถูกพัฒนาขึ้นเพื่อเพิ่มประสิทธิภาพ โดยจะมีการนำเอา U_{stl} และ U_{df} เข้ามาทดลอง และจับคู่ เพื่อทดลองการจัดงานในทุกขนาดข้อมูล ดังที่แสดงในตารางที่ 3.4 โดยทดลอง 100 รอบ ทุกชนิดข้อมูล ซึ่งจะมีการดูผลจาก m และ r ในทุกเครื่อง โดยจะมีการเก็บสถิติและเปรียบเทียบเวลาที่ใช้ในการจัดเรียงข้อมูลเปรียบเทียบกับ BQSort และ MWSort

ตารางที่ 3.4 พารามิเตอร์ที่เลือกใช้ในการทดลอง U_{stl} : U_{df} : m : r เมื่อ $B=0.01$ จากการทดลอง 100 ครั้ง

ระบบ	i7-2600	X5670	R7-1700	R9-2920
Uint32				
BF-DF	LPF	SPF	SPF	SPF
N=200M	0.5:2:2:8	0.5:1:1:6	0.5:1:2:16	0.5:1:2:12
N=500M	0.5:2:2:8	1:4:1:6	1:2:2:16	1:2:2:12
N=1000M	0.5:2:2:8	1:8:1:6	2:4:2:16	1:4:2:12
N=2000M	4:8:2:8	2:16:1:6	2:4:2:16	2:4:2:12
Uint64/Double				
BF-DF	RAL	RAL	LAL	LAL
N=200M	1:8:2:8	2:4:1:6	1:2:2:16	0.5:2:1:8
N=500M	1:8:2:8	2:4:1:6	1:2:2:16	1:4:1:8
N=1000M	2:8:2:8	4:8:1:6	2:2:2:16	2:8:1:8
N=2000M	2:8:2:8	4:8:1:6	2:2:2:16	2:8:1:8

ตารางที่ ก.1 – ก.12 แสดงเวลาที่ใช้ในการจัดเรียงข้อมูล 100 ครั้ง โดยจะแสดงอัลกอริทึมที่ใช้ ตัวชี้วัด ขนาดข้อมูล ในแต่ละชนิดข้อมูล ซึ่ง KPI ที่ใช้จะใช้ $\bar{T}$ และ σ_T เพื่อดูความเสถียรของอัลกอริทึม

3.5.4.1 การเรียงข้อมูลกับค่า U_{df}

ในการจัดงานโดยใช้ BF-DF ในรูปแบบทิศทาง (RAL และ LAL) เปรียบเทียบกับ ขนาดของข้อมูลที่คงเหลืออยู่ (SPF และ LPF) จะพบว่า SPF และ LPF จะดีกับชนิดข้อมูลที่มีขนาดเล็กเช่น Uint32 ซึ่งเราจะพบว่าค่าส่วนใหญ่จะเป็นค่าที่มีขนาดเล็กไม่ว่าจะเป็นค่าของ U_{stl} และ U_{df} ในทางกลับกัน หากขนาดของชนิดข้อมูลใหญ่ขึ้น ไม่ว่าจะเป็น Uint64 และ Double ค่า U_{stl} และ U_{df} จะมีค่าที่ใหญ่มากขึ้น เมื่อเปรียบเทียบกับ Uint32

ในภาพที่ 3.1-3.8 จะเห็นว่าทุกเครื่องที่ใช้การทดลอง มีผลการทดลองในทิศทางเดียวกันคือ ในภาพที่ 3.1-3.4 ค่า $\bar{T}_{100M}$ เพิ่มขึ้นอย่างชัดเจนตามค่า N ซึ่งต่างกับค่า $\bar{T}_{100M}$ ของการแบ่งข้อมูล และการจัดเรียงข้อมูลเพียงอย่างเดียวที่มีค่าคงที่ของแต่ละ ขนาดข้อมูล ใน U_{stl} เดียวกัน นั้นแสดงให้เห็นว่า การจัดเรียงข้อมูลนั้นเป็นกระบวนการที่จะต้องทำให้ข้อมูลมีขนาดเล็กลง ก่อนที่จะจัดเรียงข้อมูล

ในการลดเวลาที่ใช้ในการจัดเรียงข้อมูล จะต้องมีการปรับค่า U_{df} ของ การจัดงานแบบ BF-DF ซึ่ง หากเพิ่มขนาดของ U_{df} จะลดจำนวนขั้นของการวนซ้ำ ซึ่งตามภาพที่ 3.1-3.4 จะเห็นว่า $\bar{T}_{100M}$ จะมีค่าสูงขึ้น 2 เท่า ตามค่า N ค่า U_{stl} จะมีรูปแบบเช่นเดียวกันกับ U_{df} ซึ่งจะเห็นว่า U_{stl} จะมี

เอกสารนี้เป็นเอกสารที่สงวนลิขสิทธิ์สำหรับการใช้งานเพื่อการศึกษาเท่านั้น เมื่อผู้ใช้ได้เห็นใบเซอร์ยืนยันการคัดลอก
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

ความสัมพันธ์กันกับ U_{df} ซึ่งเกิดจาก T_{100M} จะมีค่ามากขึ้นในอัตราที่น้อยลง ตามภาพที่ 3.5-3.8 ดังนั้นการจัดเรียงข้อมูลที่มีขนาดใหญ่จะทำให้ใช้เวลาในการจัดเรียงข้อมูลมากกว่าการแบ่งข้อมูลและการจัดเรียงอาร์เรย์ที่มีขนาดเล็กกว่า

3.5.4.2 สถาปัตยกรรมของหน่วยความจำหลัก

ในการเปรียบเทียบ BQSort กับ MSPSort จะเห็นว่า MSPSort ใช้เวลาในการจัดเรียงข้อมูลน้อยกว่า ในทุกชนิดข้อมูล และทุกเครื่อง ยกเว้น X5670 สิ่งนี้เนื่องจาก BQSort มีอัลกอริธึมที่สามารถดึงงานมาจากแกนซีพียูอื่นได้ ซึ่งทำให้ BQSort สามารถทำงานได้ดีบนเครื่องหลายซ็อกเก็ต ที่เป็นสถาปัตยกรรม NUMA ได้ดีกว่า MSPSort

โดยใน MWSort ไม่ได้ทดลองบน $N=2000M$ ชนิดข้อมูล Uint64 และ Double บน X5670 เนื่องจากขนาดของหน่วยความจำกายภาพของเครื่องดังกล่าวมีเพียง 24 GB ซึ่งไม่เพียงพอทำให้ MWSort นั้นสามารถจัดเรียงข้อมูลได้เวลาน้อยกว่า และมีค่า σ_T น้อยที่สุด ในทุกขนาดข้อมูล สืบเนื่องจากความสมดุลของการเข้าถึงหน่วยความจำ เครื่อง X5670 และ R9-2920 มีลักษณะที่เป็นสถาปัตยกรรม NUMA ซึ่งมีช่องสัญญาณหน่วยความจำ 4 ช่อง (Channel) ในขณะที่เครื่องอื่นที่ใช้ในการทดลองมีเพียง 2 ช่อง และรองรับการทำงานหน่วยความจำที่สูง สิ่งนี้เป็นสิ่งที่แตกต่างกับเวลาที่ใช้ในการจัดเรียงข้อมูล

3.5.4.3 เสถียรภาพในการจัดเรียงข้อมูล

จะเห็นว่าเวลาที่ใช้ในการจัดเรียงข้อมูลทุกชนิดข้อมูลของทุกเครื่องนั้นจะมีค่า T มากกว่า T_{Q2} ซึ่ง σ_T และ T_{QR} เป็นค่าที่น่าสนใจเช่นกัน เนื่องจากค่าทั้งสองนี้ของ MSPSort มีค่าน้อยกว่า BQSort และ MWSort ยกเว้น X5670 ซึ่งสิ่งนี้บ่งบอกถึงความเสถียรในการจัดเรียงข้อมูลของ MSPSort

บทที่ 4

การออกแบบอัลกอริธึมการจัดเรียงแบบขนานโดยใช้หลาย สแต็ก

อัลกอริธึมการจัดเรียงแบบขนานโดยใช้หลายสแต็กได้ออกแบบให้มีความทำงานแบบขนาน ซึ่งอัลกอริธึมนี้มีหลายตัวแปรที่ส่งผลกระทบต่อประสิทธิภาพการทำงาน เช่น ขนาดบล็อก B การจัดการข้อมูลที่เหลืออยู่หลังจากแบ่งข้อมูล การจัดการงานระหว่างข้อมูลด้านซ้ายและด้านขวา

การแบ่งข้อมูลแบบ MSPPartition และการจัดเรียงข้อมูลโดยใช้ MSPSort ในบทที่ 3 สามารถอธิบายได้โดยคำนึงถึงปัจจัยต่อไปนี้ ขนาดบล็อก B การแก้ไขปัญหาตำแหน่งของไพวอทจำนวนเรดในการแบ่งข้อมูลขั้นแรก ขั้นถัดลงมา และขั้นสุดท้ายของการทำรีเคอร์ซีฟ โดยสามารถแก้ไขปัญหาและอธิบายอัลกอริธึม

4.1 ขนาดบล็อก B (หน่วยเป็นกิโลไบต์)

ภายในฟังก์ชัน MSPPartition แบ่งออกเป็น 2 ส่วนคือ Parallel Stacked Blocks Partition Step และ Unfinished Blocks Partition Step ซึ่งใน Parallel Stacked Blocks Partition Step นั้นจะเป็นการแบ่งอาร์เรย์ $A = A[0], A[1], A[2], \dots, A[N-1]$ โดยจะแบ่งข้อมูลโดยหาตำแหน่งไพวอท p และจะตัดข้อมูลออกเป็นบล็อก เพื่อให้สามารถแบ่งข้อมูลแบบขนานได้เป็นจำนวน $b = B/\text{sizeof}(\text{Type})$ ตัว จากทั้งด้านซ้ายและด้านขวา (อัลกอริธึม 4.1 บรรทัดที่ 4) และจากขวาไปซ้ายในบรรทัดที่ 7 เมื่อขนาดบล็อก B มีหน่วยเป็นกิโลไบต์ บล็อกข้อมูลจากด้านซ้ายไปขวาจะแจกให้กับเรด จำนวน τ เรด แบบรอตโรบิน (round-robin) เพื่อให้ทุกเรดเข้าถึงข้อมูลภายในบล็อกเหล่านั้นแบบขนานได้ และป้องกันการใช้ข้อมูลซ้ำกัน แต่ละเรดจะทำเปรียบเทียบข้อมูลแบบ Hoare โดยไม่เกี่ยวข้องกัน โดยถือว่าเป็น Data Parallelism

อัลกอริธึม 4.1 Optimized Multi-Stack Partition Algorithm ที่ได้รับการปรับปรุง

```

1 Function MSPPartition( $A, i_L, j_R, p, \tau$ )
2    $N_L = (p - i_L)/b$ 
3    $N_R = (j_R - p)/b$ 
4   For  $i = 0$  to  $N_L-1, i++$  do
5      $L_s[i \bmod \tau].push(i_L + i, i_L + i + b)$ 
6   End
7   For  $i = 0$  to  $N_R-1, i++$  do
8      $R_s[i \bmod \tau].push(j_R - i - b, j_R - i)$ 
9   End
10  OpenMP parallel for private( $i, j, l_b, r_b$ )
11  For  $t = 0$  to  $\tau - 1$  do
12    While  $L_s[t]$  not empty AND  $R_s[t]$  not empty do
13       $(i, l_b) = L_s[t].pop()$ 
14       $(r_b, j) = R_s[t].pop()$ 
15      Do
16        While  $A[i] \leq A[p]$  AND  $i \leq l_b$  do
17           $i = i + 1$ 
18        End
19        While  $A[j] > A[p]$  AND  $j \geq r_b$  do
20           $j = j - 1$ 
21        End
22        If  $i \leq l_b$  AND  $j \geq r_b$  then
23          SWAP( $A[i++], A[j--]$ )
24        End
25      While  $i \leq l_b$  AND  $j \geq r_b$ 
26      If  $i > l_b$  then
27         $R_s[t].push(r_b, j)$ 
28      End
29      Else if  $j < r_b$  then
30         $L_s[t].push(i, l_b)$ 
31      End

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

อัลกอริธึม 4.1 Optimized Multi-Stack Partition Algorithm ที่ได้รับการปรับปรุง (ต่อ)

```

32      End
33       $(i_L, j_R) = P_s.pop()$ 
34  End
35   $l_{min} = \min(L_s[t], \forall t)$ 
36   $r_{max} = \max(R_s[t], \forall t)$ 
37   $\mu = \text{GetMu}(i_L, j_R, r_{max}, l_{min}, u_{sort}, \tau)$ 
38   $\pi = \text{GetIndex}(i_L, j_R, r_{max}, l_{min}, u_{sort}, \tau)$ 
39  SWAP(A[p], A[ $\pi$ ])
40  If  $\mu > 1$  then
41      return MSPPartition(A,  $l_{min}, r_{max}, \pi, \mu$ )
42  End
43  Else
44      return LomutoPartition(A,  $l_{min}, r_{max}, \pi$ )
45  End
46 End Function

```

4.2 การแก้ปัญหาไฟวอทเอียงใน MSPPartition

ในการจัดเรียงข้อมูลนี้จะมีปัญหาที่เกิดขึ้น คือการแบ่งงานในแต่ละเรด ให้มีความสมดุลกัน เพื่อที่จะให้เกิดการทำงานแบบขนานมากที่สุด และแต่ละเรดจะไม่ต้องรอซึ่งกันและกัน แต่ละเรดนั้นจะมีตัวแปรโลคอลที่เป็นของตนเองตั้งตัวอย่าง i และ j เป็นตัวแปรชี้ตำแหน่งในการเปรียบเทียบทั้งด้านซ้ายและขวา ในการทำอัลกอริธึม Hoare และมีตัวแปร l_b และ r_b ในการชี้ตำแหน่งสุดท้ายในการเปรียบเทียบข้อมูลในแต่ละบล็อกทั้งด้านซ้าย และด้านขวา ถัดจากนั้น หากมีบล็อกใดที่ไม่สามารถเปรียบเทียบและสลับข้อมูลได้เสร็จสมบูรณ์ จะมีการพุนข้อมูลกลับไปยังสแต็ก ตามอัลกอริธึมที่ 4.1 บรรทัดที่ 27 และ 30 โดยการวนซ้ำนี้จะทำจนกระทั่งสแต็กข้างซ้ายหรือสแต็กข้างขวาของทุกเรดว่างเปล่า ดังอัลกอริธึม 4.1 บรรทัดที่ 12

หลังจากนั้นจะมีการตรวจสอบขอบซ้ายสุด และขอบขวาสุดของบล็อกด้านซ้ายและด้านขวาตามลำดับดังนี้ $l_{min} = \min(L_s[t], \forall t)$ และ $r_{max} = \max(R_s[t], \forall t)$ โดยที่จะเปรียบเทียบในทุกเรด จากนั้นจะมีการหาจำนวนเรดทั้งหมดโดยการเรียกฟังก์ชัน $\mu = \text{GetMu}()$ ดังอัลกอริธึม 4.1 บรรทัดที่ 36 โดยค่า μ นั้นจะนำไปเปรียบเทียบ $r_{max} - l_{min}$ โดยคำนึงถึงความยาวที่ได้จะต้องมากกว่าค่าของ u_{sort} ดังสมการ 4.1

$$\mu = \begin{cases} \tau, & \mu' \geq \tau \\ \mu', & \mu' < \tau \end{cases} \quad (4.1)$$

เมื่อ

$$\mu' = \left(\frac{r_{max} - l_{min} + 1}{u_{sort}} \right) \quad (4.2)$$

จากนั้นใน Unfinished Blocks Partition Step จะมีการแก้ปัญหาไพวอทเพียง โดยขั้นตอนนี้จะมีการเลือกไพวอทโดยใช้ฟังก์ชัน MO5() ในการทำ MSPSort ในบทที่ 3 (อัลกอริทึมที่ 3.1 บรรทัดที่ 5) จำนวนบล็อกทั้งด้านซ้ายและขวาจะมีค่าเท่ากันคือ $\mu = r_{max} - l_{min} / 2$ ซึ่งผลลัพธ์ที่ได้นี้จะเกิดปัญหาค่าไพวอทเพียงหลังจากการเรียกใช้ฟังก์ชันแบบริเคอร์ชันทุกครั้ง ในการออกแบบ MSPSort ค่า μ สามารถใช้ในการปรับตำแหน่งของไพวอท เพื่อให้จำนวนชั้นของการทำรีเคอร์ชันลดลง ดังสมการที่ 4.3

$$\pi = l_{min} + \frac{(l_{min} - i_L) \times (r_{max} - l_{min})}{(l_{min} - i_L + j_R - r_{max})} \quad (4.3)$$

เมื่อ $l_{min} - i_L$ และ $j_R - r_{max}$ เป็นความยาวของข้อมูลที่ถูกสลับแล้วด้านซ้ายและด้านขวาตามลำดับ และ $r_{max} - l_{min}$ คือความยาวของข้อมูลที่ยังไม่ได้มีการเปรียบเทียบ (Unfinished)

โดยสรุปแล้วคือ ข้อมูลที่ยังไม่มีการเปรียบเทียบมีความยาวมากจะแสดงให้เห็นว่าไพวอทมีโอกาสเอียงมากขึ้น ในการคาดเดาตำแหน่งของไพวอทสามารถใช้ผ่านสมการที่ 4.3 (อัลกอริทึม 4.1 บรรทัดที่ 40) ในขั้นตอน Unfinished Blocks Partition Step หาก $r_{max} - l_{min}$ มีขนาดสั้นกว่า u_{sort} จะมีการแบ่งข้อมูลแบบ Lomuto's และจะส่งค่าตำแหน่งของไพวอทใหม่ ดังอัลกอริทึม 4.1 บรรทัดที่ 43

4.3 อัตราส่วนเธรดในการแบ่งข้อมูล θ (Partition Thread Ratio)

จากบทที่ 3 จะเห็นว่าการที่จะแบ่งข้อมูลให้เร็วจะต้องตั้งค่า $\tau < \tau_{\max}$ โดยจะมีการตั้งค่าเริ่มต้นของเธรดในการแบ่งข้อมูลครั้งแรกโดยการนำ $\theta \times \tau_{\max}$ เมื่อ $\theta < 1.00$ โดยค่า θ จะเป็นค่าของอัตราส่วนเธรดในการแบ่งข้อมูล โดยในอัลกอริธึม 4.2 บรรทัดที่ 2 จะเห็นว่า $\theta < 1.00$ จะส่งผลต่อแคชขั้นสุดท้าย และ TLB ซึ่งเธรดที่กำลังแบ่งข้อมูลอยู่จะต้องเข้าถึงข้อมูลในหน่วยความจำหลายตำแหน่งภายในช่วงเวลานั้น

อัลกอริธึม 4.2 MSPSort ที่ปรับปรุงใหม่

```

1 Function Main()
2     MSPSort(A, 0, N-1,  $\theta \times \tau_{\max}$ )
3 End Function
4 Function MSPSort(A,  $i_L, j_R, \tau$ )
5     p = MO5(A,  $i_L, j_R$ )
6     If  $j_R - i_L > u_{sort}$  then
7         p = MSPPartition(A,  $i_L, j_R, p, \tau$ )
8         If  $\tau/\alpha > \tau_{\min}$  then
9              $\tau = \tau/\alpha$ 
10        End
11    Else
12         $\tau = \tau_{\min}$ 
13    End
14    If  $j_R - i_L > u_{trav}$  then
15        BF-MSPSort(A,  $i_L, j_R, p, \tau$ )
16    End
17    Else
18        DF-MSPSort(A,  $i_L, j_R, p, \tau$ )
19    End
20 End
21 Else
22     OpenMP Task untied
23     std::sort(A+ $i_L$ , A+ $j_R$ )

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

อัลกอริธึม 4.2 MSPSort ที่ปรับปรุงใหม่ (ต่อ)

24 End

25 End Function

4.4 การจัดวางเธรดแบบพลวัต (Dynamic Thread Allocation)

การจัดวางเธรดแบบพลวัต ประกอบไปด้วย 2 พารามิเตอร์ที่เป็นอิสระจากกันคือค่าลดทอนเธรด α และค่าทวีคูณเธรด β เมื่อ $1 < \alpha < 3$ ซึ่ง β นั้นจะเป็นตัวหารจำนวนเธรดในแต่ละชั้นของรีเคอร์ซีฟ ค่า β จะมีความสัมพันธ์กับการจัดวางตำแหน่งของไพวอท ซึ่งการที่อาร์เรย์มีขนาดใหญ่ จำเป็นต้องมีจำนวนเธรดที่มากพอ โดยหาก $\beta=ON$ แต่ละงานจะได้จำนวนเธรดที่แตกต่างกัน $\tau_L = \beta_L \times \tau$ และ $\tau_R = \beta_R \times \tau$ ตามลำดับ หาก $\beta=OFF$ ค่า $\tau_L = \tau_R = \tau$ ดังสมการที่ 4.4 และ 4.5

$$\beta_L = \begin{cases} 1 + \frac{1.5}{l+r} & , l/r \geq 3 \\ 1 & , 1 \leq l/r < 3 \\ 1 & , \text{กรณีอื่น} \end{cases} \quad (4.4)$$

และ

$$\beta_R = \begin{cases} 1 + \frac{1.5}{r-l} & , r/l \geq 3 \\ 1 & , 1 \leq r/l < 3 \\ 1 & , \text{กรณีอื่น} \end{cases} \quad (4.5)$$

4.5 อัลกอริธึม Depth First Traversal

ในอัลกอริธึม MSPSort() จะมีการเรียกใช้ MO5() ในอัลกอริธึม 4.2 บรรทัดที่ 5 เพื่อหาค่าไพวอท ที่ตำแหน่ง p จากนั้นจะมีการสลับกับตำแหน่งกลางของอาร์เรย์ย่อย A จากนั้นจะมีการเรียกใช้ MSPPartition() เพื่อแบ่งข้อมูลด้วยไพวอทนั้น และส่งค่าตำแหน่งใหม่ของไพวอทกลับมา ดังอัลกอริธึม 4.2 บรรทัดที่ 7 จากนั้นอัลกอริธึมการจัดเรียงข้อมูลจะเรียกฟังก์ชันซ้ำแบบ BF ดังอัลกอริธึม 4.2 บรรทัดที่ 15 โดยรายละเอียดของ BF-MSPSort() อธิบายในอัลกอริธึม 4.3 ซึ่งจะเห็นได้ว่า MSPSort() จะเรียกใช้ OpenMP Task 2 ครั้ง ที่มีการเรียกฟังก์ชันแบบรีเคอร์ซีฟ ตามอัลกอริธึม 4.3 บรรทัดที่ 3 และ 7 โดยบรรทัดที่ 3 คือด้านซ้าย และบรรทัดที่ 7 คือด้านขวา

อัลกอริธึม 4.3 MSPSort ชนิด Breadth-First และชนิด Depth-First

```

1 Function BF-MSPSort(A,  $i_L$ ,  $j_R$ , p,  $\tau$ )
2      $\tau_L = \beta_L \times \tau$ 
3     OpenMP Task untied
4     MSPSort(A,  $i_L$ , p-1,  $\tau_L$ )
5      $\tau_R = \beta_R \times \tau$ 
6     OpenMP Task untied
7     MSPSort(A, p+1,  $j_R$ ,  $\tau_R$ )
8 End Function

9 Function DF-MSPSort(A,  $i_L$ ,  $j_R$ , p,  $\tau$ )
10     $P_s$ .push( $i_L$ ,  $j_R$ )
11    While  $P_s$  IS NOT empty do
12        ( $i_L$ ,  $j_R$ ) =  $P_s$ .pop()
13        If  $j_R - i_L > u_{sort}$  then
14            p = MO5(A,  $i_L$ ,  $j_R$ )
15            p = MSPPartition(A,  $i_L$ ,  $j_R$ , p,  $\tau$ )
16             $P_s$ .push( $i_L$ , p-1)
17             $P_s$ .push(p+1,  $j_R$ )
18        End
19    Else
20        OpenMP Task untied
21        std::sort(A+ $i_L$ , A+ $j_R$ )
22    End
23 End
24 End Function

```

โดยปกติแล้วอัลกอริธึม MSPSort() จะมีการจัดลำดับงาน (Scheduling) แบบ BF เท่านั้น ซึ่งผลลัพธ์ที่ได้คือ จะมีการจัดเรียงข้อมูลเมื่อขนาดของอาร์เรย์เล็กกว่า u_{sort} โดยการแบ่งเรดออกไปโดยไม่เกี่ยวข้องกับเรดก่อนหน้า ดังอัลกอริธึม 4.2 บรรทัดที่ 23 โดยใช้ std::sort เมื่อความยาวคัตออฟการจัดเรียง (Sorting Cutoff) คือ $u_{sort} = U_{sort}/\text{sizeof}(\text{Type})$ เมื่อ Type คือชนิดของข้อมูลที่จะนำมาจัดเรียงหน่วยเป็นไบต์

ในการเพิ่มประสิทธิภาพในการทำงานแบบขนาน การจัดงานแบบ BF เป็นตัวเลือกหนึ่ง อย่างไรก็ตามการจัดงานแบบ BF นั้นไม่สามารถทราบได้ว่าอัลกอริธึมจะมีการเลือกทำงานที่บริเวณใดของข้อมูลอาเรย์ ซึ่งส่งผลให้เกิดปัญหา dTLB misses page faults และ branch/memory stalls ซึ่งโดยทั่วไปแล้วปัญหานี้สามารถแก้ไขได้โดยการปรับการแบ่งข้อมูลให้อยู่ในรูปแบบของการทำ Depth-First หรือ DF เพื่อให้ทำงานทางลึกก่อนโดยอัลกอริธึมที่จะถูกพัฒนาขึ้นคือ DF-MSPSort() โดยหากอาเรย์ $j_R - i_L$ มีค่าน้อยกว่า u_{trav} (อัลกอริธึม 4.2 บรรทัดที่ 18) MSPSort จะเปลี่ยนรูปแบบการทำงานจาก BF ไปเป็น DF โดยที่ค่า $u_{trav} = U_{trav}/sizeof(\text{Type})$ ซึ่งอัลกอริธึม 4.3 เป็นการจัดเรียงแบบ RAL (Right Always) ซึ่งเป็นตัวอย่างหนึ่งในการจัดลำดับงานแบบ BF-DF

การ Traversed BF-DF จะสามารถปรับได้ 2 รูปแบบคือตามทิศทาง และตามขนาดของข้อมูล โดยทิศทางจะมี RAL และ LAL (Left Always) โดย RAL จะพหุข้อมูลด้านซ้ายลงในสแต็กก่อนแล้วตามด้วยด้านขวา จากนั้นจึงป๊อปด้านขวาออกมา เพื่อให้เลือกข้อมูลด้านขวามาทำก่อน ในทางกลับกัน LAL จะพหุข้อมูลด้านขวาลงในสแต็กก่อนแล้วตามด้วยด้านซ้าย จากนั้นป๊อปออกมา เพื่อให้เลือกข้อมูลด้านซ้ายทำก่อน นอกจาก BF-DF Traversed โดยอาศัยรูปแบบทิศทางแล้ว ยังมีรูปแบบที่ตรวจสอบขนาดของอาเรย์ย่อยคือแบบความยาวมากทำก่อน (Longer Partition First: LPF) และความยาวอาเรย์ย่อยสั้นกว่าทำก่อน (Shorter Partition First: SPF) โดยจะตรวจสอบขนาดของอาเรย์ย่อย จากนั้นจึงเลือกขอบเขตข้อมูลพหุลงสแต็ก

4.6 ค่าเรดต่ำสุด $\tau_{\min}$

โดยปกติแล้วอัลกอริธึมที่นำเสนอจะมีจำนวนของซอฟต์แวร์เรด τ ซึ่งจะมีจำนวนลดลงไปเป็น τ/α ดังอัลกอริธึม 4.2 บรรทัดที่ 12 และมีค่า $\tau = \tau_{\min}$ เมื่อ $\tau_{\min}$ เป็นค่าเรดต่ำสุดที่จะถูกใช้งานในรอบการแบ่งข้อมูลนั้น เนื่องจากข้อมูลที่ถูกแบ่งจะมีจำนวนอาเรย์ย่อยมากขึ้นแบบเอกซ์โพเนนเชียลเมื่อมีการทำรีเคอร์ซีฟ ในขณะที่จำนวนของแกนซีพียูยังคงมีเท่าเดิม ทำให้ปริมาณแกนซีพียูไม่เพียงพอกับจำนวนอาเรย์ย่อยที่เพิ่มมากขึ้น จึงต้องมีการปรับลดจำนวนเรด เพื่อให้มีความเหมาะสมกับจำนวนแกนซีพียูที่มีอยู่อย่างจำกัด และจัดการการจราจรของหน่วยความจำให้เหมาะสม

4.7 การทดลอง และตัวชี้วัด

การทดลองอัลกอริธึมการจัดเรียงข้อมูลทั้งหมด ทำบนเครื่องลินุกซ์ 2 ชนิด เพื่อทดสอบประสิทธิภาพ อัลกอริธึมที่ผ่านการออปติไมซ์ แบ่งเป็นเครื่องที่มีขนาดเล็กและมีขนาดใหญ่ ซึ่งจะมีการแสดงผลลัพธ์ ในหัวข้อถัดไป

4.7.1 เครื่องที่ใช้ในการทดลอง

ตารางที่ 4.1 ข้อมูลของเครื่องที่ใช้ในการทดลอง

หน่วยประมวลผลกลาง	Cortex A72 Pi4	Thread Ripper R9-2920
สัญญาณนาฬิกา (GHz)	1.50	3.50
c (แกนประมวลผล)	4	24
หน่วยความจำหลัก	8 GB	64 GB
การติดตั้ง	1 x 8 GB	8 x 8 GB
เทคโนโลยี	DDR4	DDR4
ความถี่ (MHz)	2133	2400
L1 I-Cache	4x48KB 3 way	12x64KB 4 way
L1 D-Cache	4x32KB 2 way	12 x32KB 8 way
L2 Cache	1024KB 16 way	12 x 512KB 8 way
L3 Cache	-	4x 8MB 16way
ระบบปฏิบัติการ	Raspberry Pi	Ubuntu
รุ่น	Beta	18.04 LTS
เคอร์เนล (Kernel)	5.4.42-v8+	5.3.0-64-generic
คอมไพเลอร์ (G++)	8.3.0	7.5.0
โอเพนเอ็มพี	4.5	4.5

อัลกอริธึม MSPSort ที่ได้รับการออปติไมซ์จะมีการทดลองในเครื่อง 2 ชนิด ดังตารางที่ 4.1 โดยอัลกอริธึมที่ทดลองนั้นจะคอมไพล์ด้วย G++ 8.3.0 บนเครื่อง Cortex A72 Pi4 และ GCC 7.5.0 บนเครื่อง AMD Thread Ripper R9-2920 โดยเครื่อง Pi4 จะมีความแตกต่างจากเครื่อง R9-2920 ในด้านจำนวนแกนซีพียู ที่มีความถี่สัญญาณนาฬิกาต่ำและสูงตามลำดับ ขนาดของหน่วยความจำหลัก โครงสร้างของแคชที่แตกต่างกัน โดย Raspberry Pi4 จะไม่มีแคชระดับที่ 3 ซึ่งสิ่งนี้เป็นที่น่าสนใจว่า Raspberry Pi4 เป็นตัวแทนของเครื่องทั่วไป ในขณะที่ AMD Thread Ripper แทนเครื่องที่มีประสิทธิภาพสูง

ตารางที่ 4.2 แสดงพารามิเตอร์ที่ใช้ในการทดลองของบทที่ 4 โดยจะมีการทดลองอัลกอริธึม MSPSort และใช้ BQSort และ MWSort ในการเปรียบเทียบ ในการจัดเรียงข้อมูลรูปแบบ Uniform เอกสารนี้เป็นเอกสารที่สงวนลิขสิทธิ์ไว้เพื่อใช้ในการศึกษาเท่านั้น เมื่อผู้ใช้ได้เห็นเอกสารนี้แล้วไม่ควรนำเอกสารนี้ไปเผยแพร่หรือใช้เพื่อวัตถุประสงค์อื่นใดโดยไม่ได้รับอนุญาตจากผู้จัดทำเอกสารนี้

ทั้งชนิดข้อมูล Uint32 และ Double มีการออปติไมเซชันของ GCC เป็นแบบ O2 เช่นเดียวกันกับบทที่ 3 เพื่อลดเวลาในการทดลองจัดเรียงข้อมูล โดยในการทดลองนี้จะมีการทดลองบนเครื่อง Pi4 ด้วยข้อมูลขนาด 20M ถึง 200M ตัวและทดลองบน R9 ด้วยขนาดข้อมูล 200M ถึง 2000M ตัว

ตารางที่ 4.2 พารามิเตอร์ที่ใช้ในการทดลองของ MSPSort, BF: Bread-First, DF: Depth-First

พารามิเตอร์	ค่าที่ใช้
อัลกอริธึม	MSPSort, BQSort, MWSort
ชนิดข้อมูล	Uint32, Double
การสุ่มข้อมูล	Uniform
การออปติไมเซชันของ GCC	O2
ขนาดข้อมูล Pi4	20M, 50M, 100M, 200M
ขนาดข้อมูล R9	200M, 500M, 1000M, 2000M
การจัดงาน	RAL, LAL, LPF, SPF
ขนาดบล็อก B	32, 64, 128, 256 KB
ค่าการจัดเรียง U_{sort}	2, 4, 8, 16 MB
ค่าการจัดงาน U_{trav}	8, 16, 32, 64 MB
ตัวคูณเรด m	1, 2
เรดขั้นต่ำ τ_{min}	2, 3
อัตราส่วนการแบ่งงานของเรด θ	0.67, 0.75, 0.8, 0.9
ตัวหารเรด α	4/3, 3/2, 2 5/2

สิ่งที่แตกต่างจากบทที่ 3 คือขนาดบล็อก B หน่วยเป็น KB ค่าของ U_{sort} หน่วยเป็น MB และ U_{trav} หน่วยเป็น MB เช่นเดียวกัน ซึ่งมีค่าพารามิเตอร์ที่ใช้ในการทดลองดังตารางที่ 4.2 ซึ่งในการทดลองนี้ไม่ได้อ้างอิงจากขนาดของแคชระดับที่ 3 เนื่องจาก Pi4 ไม่มีแคชระดับที่ 3 ดังที่กล่าวไว้ข้างต้น ในบทนี้ MSPSort จะสร้างจำนวนเรด τ_{max} เป็นจำนวน $m \times c$ เรด เมื่อค่าตัวคูณ $m = 1$ และ 2 ค่า τ_{min} จะมีค่าเป็น 2 และ 3

4.7.2 ตัวชี้วัดในการทดลอง

ในบทนี้มีการทดลองโดยใช้ตัวชี้วัดหลายตัวเพื่อให้เห็นมิติที่หลากหลายมากขึ้น โดยตัวชี้วัดแต่ละตัวจะเป็นตัวที่ใช้ในการวัดประสิทธิภาพของอัลกอริธึมการจัดเรียงข้อมูล โดยมี ค่าเฉลี่ยเวลาที่ใช้ในการจัดเรียงข้อมูล $\bar{T}$ และค่าเบี่ยงเบนมาตรฐานในการจัดเรียงข้อมูล σ_T เป็นตัวที่ใช้ในการวัดเวลาใน

การจัดเรียงข้อมูล และเสถียรภาพในการทำงานของอัลกอริธึม และมีตัวชี้วัดอื่นที่ใช้ในการวัดประสิทธิภาพของอัลกอริธึมดังนี้

4.7.2.1 ค่าเฉลี่ยเวลาที่ใช้ในการจัดเรียงข้อมูล ($\bar{T}$) และเวลาที่ใช้ต่อข้อมูล z ($\bar{T}_z$) ล้านตัว

ในงานนี้จะมีการใช้ `omp_get_wtime()` ซึ่งเป็นฟังก์ชันในการวัดค่าเวลาของอัลกอริธึม โดยจะมีการเรียกใช้ 2 ครั้ง ครั้งที่ 1 เรียกใช้ก่อนการจัดเรียงข้อมูล และครั้งที่ 2 หลังจากจัดเรียงข้อมูลเสร็จแล้ว ซึ่งค่าเวลานี้ จะไม่รวมระยะเวลาในการสร้างข้อมูลตั้งต้นในอาเรย์

ค่าเฉลี่ยเวลาที่ใช้ในการจัดเรียงข้อมูล z ล้านตัว เรียกว่า เวลาต่อข้อมูล z ล้านตัว ($\bar{T}_z$) ตัวชี้วัดนี้สามารถนำเอาผลลัพธ์มาเปรียบเทียบกับแต่ละข้อมูลขนาดเท่ากัน (z ล้านตัว) ในทุกการทดลอง ทำให้สามารถเปรียบเทียบข้อมูลกับทุกเครื่องที่ใช้ในการทดลองที่จัดเรียงข้อมูลด้วยอัลกอริธึมเดียวกัน

4.7.2.2 ส่วนเบี่ยงเบนมาตรฐานของเวลา σ_T และ σ_z

ในการทดลองนี้จะมีค่าส่วนเบี่ยงเบนมาตรฐานของเวลา σ_T ซึ่งเป็นค่าที่แสดงถึงเสถียรภาพของอัลกอริธึม เนื่องจากข้อมูลที่ใช้ในการทดลองเป็นข้อมูลที่มีการสุ่ม นอกจากนั้นการนำเอาส่วนเบี่ยงเบนมาตรฐานมาหาอัตราส่วนต่อ z ล้านตัว จะทำให้สามารถเลือกใช้ค่าขนาดบล็อก B ได้ดีขึ้น

4.7.2.3 เสถียรภาพของเวลา

นอกจากการหาค่าเสถียรของอัลกอริธึมด้วยส่วนเบี่ยงเบนมาตรฐานแล้ว ในทางสถิติสามารถหาควอไทล์ที่ 1 2 และ 3 หรือ T_{Q1} T_{Q2} และ T_{Q3} ได้ตามลำดับ และสามารถนำเอามาคำนวณหาควอไทล์ได้จาก $T_{QR} = T_{Q3} - T_{Q1}$ เพื่อวิเคราะห์ความเสถียรของอัลกอริธึมได้เช่นกัน

4.7.2.4 ร้อยละประสิทธิภาพการใช้งานซีพียู ในโหมดผู้ใช้ V_{usr}

ในการทดลองประสิทธิภาพการใช้งานของซีพียูนั้น สามารถอ่านข้อมูลดิบจาก ไฟล์ `/proc/stat` ซึ่งผลลัพธ์ที่สามารถวัดได้มี 4 โหมดหลักคือโหมด `user` `system` `nice` และ `idle` โดยในการวัดประสิทธิภาพนั้น สิ่งที่น่าสนใจคือส่วนของ `user` ซึ่งควรจะมีค่าที่สูงพอสมควร เพื่อเป็นตัวบ่งชี้ว่าอัลกอริธึมแบบขนานที่ทำงานสามารถดึงเอาซีพียูมาใช้งานได้เต็มประสิทธิภาพ

4.7.2.4 ร้อยละประสิทธิภาพการใช้งานหน่วยความจำหลัก V_{ram}

หน่วยความจำหลักเป็นทรัพยากรที่จำกัดในอุปกรณ์พกพา การวัดประสิทธิภาพการใช้งานหน่วยความจำหลักจึงเป็นสิ่งที่มีความสำคัญ คำสั่ง `/usr/bin/time` ซึ่งอยู่ภายใต้ระบบปฏิบัติการลินุกซ์ จึงถูกติดตั้งเพิ่มเติมเพื่อใช้วัดขนาดของหน่วยความจำที่ใช้ในหน่วยความจำหลักมี หน่วยเป็นกิโลไบต์ โดย

$$V_{ram}(\%) = \frac{1024 \times \text{MaxResident} \times 100}{N \times \text{type}} \quad (4.6)$$

เมื่อ MaxResident คือ ขนาดของหน่วยความจำที่ใช้ในหน่วยความจำหลัก N คือจำนวนข้อมูล

4.8 ผลการทดลองและอภิปรายผลการทดลอง

ในงานนี้จะเป็นการทำออปติไมซ์ และปรับค่าพารามิเตอร์ของ MSPSort เพื่อให้มีประสิทธิภาพสูงที่สุด โดยจะมีการปรับค่าพารามิเตอร์ดังต่อไปนี้

4.8.1 BF-MSPSort และค่าขนาดบล็อก B

ในส่วนนี้จะเป็นการปรับค่าขนาดบล็อก ของชนิดไฟล์ Uint32 และ Double โดยในบทที่ 3 ขนาดของบล็อกอยู่ที่ $B=0.01 \times 8\text{MB}$ ซึ่งมีค่าประมาณ 80KB โดยจะมีการปรับค่า B ให้มีค่าเป็นตัวคูณของ 32KB หรือขนาดเท่ากับขนาดของแคชระดับที่ 1 ของทั้ง 2 เครื่อง โดยค่าขนาดบล็อกนั้น หากเป็นชนิดตัวแปร Double มีขนาดเป็น 2 เท่าของ Uint32 ตามที่แสดงในตารางที่ 4.3-4.6

ตารางที่ 4.3 $\bar{T}_{10}$ และ σ_{10} ของ BF-MSPSort ด้วยชนิดตัวแปร Uint32 ในแต่ละขนาดบล็อก B บน Pi4 จำนวน 100 ครั้ง

B (KB)	ตัวชี้วัด (วินาที)	20M	50M	100M	200M
32	$\bar{T}_{10}$	0.6396	0.6615	0.7056	0.7332
	σ_{10}	0.0079	0.0058	0.0139	0.0068
64	$\bar{T}_{10}$	0.6311	0.6627	0.6948	0.7359
	σ_{10}	0.0112	0.0054	0.0071	0.0138
128	$\bar{T}_{10}$	0.7270	0.7616	0.7870	0.8334
	σ_{10}	0.0205	0.0139	0.0090	0.0180

ตารางที่ 4.4 $\bar{T}_{10}$ และ σ_{10} ของ BF-MSPSort ด้วยชนิดตัวแปร Double ในแต่ละขนาดบล็อก B บน Pi4 จำนวน 100 ครั้ง

B (KB)	ตัวชี้วัด (วินาที)	20M	50M	100M	200M
64	$\bar{T}_{10}$	1.0483	1.0978	1.1359	1.2004
	σ_{10}	0.0250	0.0309	0.0193	0.0217
128	$\bar{T}_{10}$	1.0714	1.0887	1.1576	1.1986
	σ_{10}	0.0203	0.0207	0.0383	0.0245
256	$\bar{T}_{10}$	1.0253	1.1047	1.1560	1.2007
	σ_{10}	0.0115	0.0298	0.0379	0.0249

ค่า B ที่ถูกเลือกของ Uint32 คือ 64KB และ Double ที่ 128 KB ของทั้ง 2 เครื่อง ซึ่งหากพิจารณาที่เครื่อง Pi4 ที่ N=200M ตัว และเครื่อง R9-2920 ที่ N = 2000M ตัว จะเห็นได้ว่าค่านี้จะเป็นค่าที่ดีที่สุดของทุกตัวชี้วัดในการทดลอง ไม่ว่าจะเป็นค่า $\bar{T}_{10}$ และ σ_{10} ของเครื่อง Pi4 และ $\bar{T}_{100}$ และ σ_{100} ของเครื่อง R9-2920

ตารางที่ 4.5 $\bar{T}_{100}$ และ σ_{100} ของ BF-MSPSort ด้วยชนิดตัวแปร Uint32 ในแต่ละขนาดบล็อก B บน R9-2920 จำนวน 100 ครั้ง

B (KB)	ตัวชี้วัด (วินาที)	20M	50M	100M	200M
32	$\bar{T}_{100}$	0.5762	0.6008	0.6244	0.6609
	σ_{100}	0.0576	0.0172	0.0206	0.0285
64	$\bar{T}_{100}$	0.5774	0.6018	0.6250	0.6533
	σ_{100}	0.0120	0.0147	0.0210	0.0141
128	$\bar{T}_{100}$	0.5753	0.5963	0.6247	0.6535
	σ_{100}	0.0081	0.0141	0.0164	0.0156

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

ตารางที่ 4.6 $\bar{T}_{100}$ และ σ_{100} ของ BF-MSPSort ด้วยชนิดตัวแปร Double ในแต่ละขนาดบล็อก B บน R9-2920 จำนวน 100 ครั้ง

B (KB)	ตัวชี้วัด (วินาที)	20M	50M	100M	200M
64	$\bar{T}_{100}$	0.8775	0.9503	0.9937	1.0484
	σ_{100}	0.0208	0.249	0.0194	0.0317
128	$\bar{T}_{100}$	0.8785	0.9473	0.9908	1.0325
	σ_{100}	0.0154	0.224	0.0179	0.0382
256	$\bar{T}_{100}$	0.8776	0.9481	0.9827	1.0397
	σ_{100}	0.0133	0.205	0.0265	0.0411

4.8.2 BF-MSPSort และค่าขนาด U_{sort} , m , τ_{min}

จากการทดลองก่อนหน้านี้หากมีการเลือกค่า B แล้ว พารามิเตอร์ถัดไปที่จะมีการปรับคือ U_{sort} , m และ τ_{min} โดยค่าที่ดีที่สุดของ m และ τ_{min} ได้แสดงบนตารางที่ 4.7 จะเห็นว่า U_{sort} จะเป็นตัวแบ่งเวลาระหว่างการแบ่งข้อมูล และเวลาที่ใช้ในการจัดเรียงข้อมูลภายหลังการแบ่งข้อมูลเสร็จแล้ว

ตารางที่ 4.7 ค่าพารามิเตอร์ (B , U_{sort} , m , τ_{min}) ที่ดีที่สุดของการจัดงานแบบ BF ในทุก N จำนวน 20 ครั้ง

ชนิดข้อมูล	Pi4	R9-2920
Uint32	(64KB, 4MB, 1, 2)	(64KB, 4MB, 2, 2)
	(64KB, 4MB, 1, 3)	(64KB, 4MB, 1, 3)
	(64KB, 4MB, 2, 2)	(64KB, 4MB, 1, 2)
Double	(128KB, 8MB, 1, 2)	(128KB, 8MB, 1, 2)
	(128KB, 8MB, 2, 2)	(128KB, 8MB, 1, 3)
	(128KB, 4MB, 1, 2)	(128KB, 4MB, 2, 2)

4.8.3 BF-MSPSort เปรียบเทียบกับการจัดลำดับงานแบบ BF-DF

ในตารางที่ 4.5 แสดงพารามิเตอร์ของ BF-MSPSort โดยในหัวข้อนี้จะเป็นการหาพารามิเตอร์ที่ดีที่สุดในการจัดงานหรือ U_{trav} ซึ่งจะต้องตั้งค่าให้เป็นตัวคูณของ 8MB ซึ่งเป็นขนาดของแคชระดับที่ 3 ในการออกแบบ U_{trav} จะต้องมีการปรับค่าให้เหมาะสม โดยจากการทดลอง RAL BF-DF เป็นวิธีการจัดการงานที่ดีที่สุดกล่าวคือมี $\bar{T}$ และ σ_T ที่ดีที่สุด โดยพารามิเตอร์ที่เลือกแสดงในตารางที่ 4.8

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

ตารางที่ 4.8 ค่าพารามิเตอร์ที่ถูกเลือก (B , U_{sort} , U_{trav} , m , τ_{min}) ที่ดีที่สุดของการจัดงานแบบ RAL-MSPSort ในทุก N จำนวน 100 ครั้ง

ขนาดข้อมูล	Uint32	Double
Pi4 20M-200M	(64KB, 4MB, 4MB, 1, 2)	(128KB, 8MB, 8MB, 1, 2)
R9-2920 200M	(64KB, 2MB, 8MB, 2, 2)	(128KB, 4MB, 8MB, 1, 2)
500M	(64KB, 4MB, 16MB, 2, 2)	(128KB, 8MB, 16MB, 1, 2)
1000M	(64KB, 8MB, 32MB, 2, 2)	(128KB, 8MB, 32MB, 1, 2)
2000M	(64KB, 8MB, 32MB, 2, 2)	(128KB, 8MB, 64MB, 1, 2)

อัลกอริธึม RAL-MSPSort เป็นอัลกอริธึมที่จัดเรียงข้อมูลและได้ผลลัพธ์ของเวลาที่ดียิ่งที่สุดเมื่อเปรียบเทียบกับ การจัดงานทุกแบบ แม้ว่าการจัดงานแบบ BF จะใกล้เคียงกับอัลกอริธึม SPF โดยอัลกอริธึมการจัดงานแบบ BF และ SPF อาศัยที่สั้นกว่าจะถูกจัดเรียงข้อมูลได้เร็วเนื่องจาก SPF จะเลือกแบ่งข้อมูลและจัดเรียงข้อมูลอาเรย์ที่มีขนาดเล็กกว่าก่อน

ในเครื่อง Pi4 ซึ่งมีแคชระดับที่ 2 แบ่งกันอยู่ระหว่างแกนซีพียูทั้ง 4 แกนนั้น RAL-MSPSort ใช้เวลาน้อยกว่าการจัดงานแบบ BF โดยเมื่อ $U_{\text{trav}} = U_{\text{sort}}$ การแบ่งข้อมูลของ MSPPartition() จะทำงานมากกว่า 1 ครั้งซึ่งทำให้อาเรย์ที่ได้สั้นกว่าค่าของ U_{sort}

ค่า U_{trav} จะมีความแตกต่างกันเนื่องจากชนิดของข้อมูล รูปแบบการจัดวางของแคชของซีพียูบนอุปกรณ์พกพา ซึ่งส่วนใหญ่แล้วซีพียูของอุปกรณ์เหล่านี้จะไม่มีแคชระดับ 3 มีสูงสุดแค่เพียงระดับที่ 2 ที่และแบ่งกันระหว่างแกนซีพียู เช่น ARM big.Little จึงอาจทำให้เกิดปัญหาแคชโคเฮเรนซ์ (cache coherence) บริเวณขอบของอาเรย์

4.8.4 RAL-MSPSort กับ อัตราส่วนเรดในการแบ่งข้อมูล θ

ในการทดลองนี้จะเป็นการหาจำนวนเรดที่ใช้ในการแบ่งข้อมูลซึ่งจะส่งผลต่อเวลาเฉลี่ยในการจัดเรียงข้อมูล ค่าเรดสูงสุดจะถูกตั้งค่าเอาไว้ที่ τ_{max} และจะลดค่าลงตามสูตร $\theta \times \tau_{\text{max}}$ โดยที่ค่า $\theta \leq 1.00$ ตามตารางที่ 4.6 โดยที่ ค่า $\bar{T}$ และ σ_T ของ RAL-MSPSort ในแต่ละ θ ซึ่งแสดงในตารางที่ 4.9 ของ Pi4 และตารางที่ 4.10 ของ R9-2920

จากตารางที่ 4.9 จะเห็นว่าเครื่อง Pi4 จะมีปัญหาด้านประสิทธิภาพทั้งข้อมูล Uint32 และ Double เมื่อมีการปรับค่า $\theta < 1.00$ ในทางกลับกันเครื่อง R9-2920 หากค่า $\theta < 1.00$ จะทำให้เวลาที่ใช้ในการจัดเรียงข้อมูลมีค่าน้อยลง ซึ่งจะเห็นได้ว่า Pi4 มีจำนวนซีพียูจำกัด เมื่อเทียบกับการเข้าถึงหน่วยความจำ

เครื่อง R9-2920 มีประสิทธิภาพของการจัดเรียงข้อมูลดีขึ้นเมื่อชนิดข้อมูลเป็น Double และ $\theta = 0.8$ ซึ่งอาจเกิดจากชนิดข้อมูล Double มีขนาดเป็น 2 เท่าของ Uint32 ซึ่งส่งผลให้แคชระดับที่ 3 มีการใช้งานเพิ่มเป็น 2 เท่าและจะต้องมีการเข้าถึงหน่วยความจำหลักมากขึ้นจากชนิดข้อมูล Uint32 นอกจากนี้สาเหตุอาจเกิดจากจำนวนฮาร์ดแวร์ที่ใช้ในการประมวลผลข้อมูลชนิด floating-point มีอยู่จำกัด ซึ่งเครื่อง R9-2920 มีแกนซีพียูจำนวน 12 แกน และ 24 เธรด ทำให้มีจำนวนฮาร์ดแวร์ในการประมวลผลข้อมูลชนิด floating-point มีจำนวนจำกัด เพราะจำนวนซอฟต์แวร์เธรดมีค่าเป็น 24

ตารางที่ 4.9 ค่า $\bar{T}$ และ σ_T ของ RAL-MSPSort ชนิดข้อมูล Uint32 และ Double ในแต่ละค่า N และค่าอัตราส่วนเธรด θ บนเครื่อง Pi4 จำนวน 100 ครั้ง

θ	KPI	20M	50M	100M	200M
Uint32 $\theta = 1.00$	$\bar{T}$	1.2640	3.2878	6.8597	14.3510
	σ_T	0.0269	0.0357	0.0805	0.1368
$\theta = 0.75$	$\bar{T}$	1.2945	3.3392	6.8744	14.6111
	σ_T	0.0544	0.0578	0.0699	0.2899
Double $\theta = 1.00$	$\bar{T}$	2.1488	5.4478	11.4136	24.2979
	σ_T	0.0491	0.1079	0.2354	0.5938
$\theta = 0.75$	$\bar{T}$	2.1637	5.5548	11.5595	24.6450
	σ_T	0.0251	0.0666	0.1432	0.2352

ตารางที่ 4.10 ค่า $\bar{T}$ และ σ_T ของ RAL-MSPSort ชนิดข้อมูล Uint32 และ Double ในแต่ละค่า N และค่าอัตราส่วนเรต θ บนเครื่อง R9-2920 จำนวน 100 ครั้ง

θ	KPI	20M	50M	100M	200M
Uint32 $\theta = 1.00$	$\bar{T}$	1.1598	2.9678	6.2311	12.5458
	σ_T	0.0146	0.0245	0.1192	0.2564
$\theta = 0.90$	$\bar{T}$	1.1459	2.9757	6.1624	12.7117
	σ_T	0.0245	0.0847	0.2307	0.3633
$\theta = 0.80$	$\bar{T}$	1.1396	2.9329	6.2100	12.9028
	σ_T	0.0167	0.0388	0.1980	0.5081
$\theta = 0.75$	$\bar{T}$	1.1429	2.9468	6.2087	12.8381
	σ_T	0.0140	0.0524	0.2250	0.4650
$\theta = 0.67$	$\bar{T}$	1.1437	2.9518	6.2619	12.9881
	σ_T	0.0223	0.0566	0.3486	0.6049
Double $\theta = 1.00$	$\bar{T}$	1.7408	4.6700	9.6917	20.1682
	σ_T	0.0231	0.1736	0.2836	0.8512
$\theta = 0.90$	$\bar{T}$	1.7363	4.6454	9.5878	20.3231
	σ_T	0.0189	0.1460	0.2413	0.5916
$\theta = 0.80$	$\bar{T}$	1.7118	4.6077	9.5261	20.0296
	σ_T	0.0795	0.2034	0.4592	1.0444
$\theta = 0.75$	$\bar{T}$	1.7230	4.6286	9.5329	20.1099
	σ_T	0.0318	0.1658	0.4191	0.7131
$\theta = 0.67$	$\bar{T}$	1.7197	4.6421	9.6873	20.4310
	σ_T	0.0384	0.1327	0.3766	0.9811

4.8.5 RAL-MSPSort กับค่าลดทอนเรต α

ค่าตัวหารเรตในบทที่ 3 มีค่าที่คงที่คือ 2 โดยในการทดลองนี้จะมีการปรับค่า α ให้มีค่าน้อยกว่า 2 โดยมีสัดส่วนเป็น $5/3$ $3/2$ และ $4/3$ เพื่อให้จำนวนเรตลดลงอย่างช้าๆ ในการทำรีเคอร์ซีฟในชั้นถัดไป ในทางกลับกัน ค่า α ที่มากกว่า 2 เช่น $5/2$ จะทำให้จำนวนเรตลดลงเร็วขึ้นเป็นการแสดงให้เห็นว่าพฤติกรรมที่เกิดขึ้นเมื่อมีจำนวนเรตที่มากขึ้น จากการทดลองก่อนหน้านี้จะนำพารามิเตอร์ที่ดีที่สุด มาทดลองพร้อมกับค่า α ดังตารางที่ 4.11 และ 4.12

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้าไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

จากผลการทดลองในตารางที่ 4.9 และ 4.10 พบว่า ค่า $\alpha = 2$ เป็นค่าที่ดีที่สุดของ Pi4 ทั้ง 2 ชนิดข้อมูล แต่ใน R9-2920 พบว่าค่าที่ดีที่สุดของ N=1000M และ 2000M ตัวคือ $\alpha = 5/3$ ซึ่งเกิดจากปริมาณงานของชนิดข้อมูล Double ที่มากขึ้น จำเป็นต้องใช้จำนวนเรดมากตาม

ตารางที่ 4.11 ค่า $\bar{T}$ และ σ_T ของ RAL-MSPSort ชนิดข้อมูล Uint32 และ Double ที่ $\theta = 1.00$ ในแต่ละค่า N, α และ β บนเครื่อง Pi4 จำนวน 100 ครั้ง

α และ β	KPI	20M	50M	100M	200M
Uint32					
$\alpha = 4/3$	$\bar{T}$	1.2931	3.3368	6.9511	14.5192
$\beta = \text{OFF}$	σ_T	0.0265	0.0568	0.0909	0.2386
$\alpha = 3/2$	$\bar{T}$	1.2931	3.3321	6.8597	14.4847
$\beta = \text{OFF}$	σ_T	0.0297	0.0638	0.0976	0.2391
$\alpha = 2$	$\bar{T}$	1.2640	3.2878	6.8597	14.3510
$\beta = \text{OFF}$	σ_T	0.0269	0.0357	0.0805	0.1368
$\alpha = 5/2$	$\bar{T}$	1.3303	3.4218	7.0489	14.7332
$\beta = \text{OFF}$	σ_T	0.0723	0.0686	0.1065	0.1512
Double					
$\alpha = 5/3$	$\bar{T}$	2.1535	5.5741	11.5349	24.4927
$\beta = \text{OFF}$	σ_T	0.0464	0.1681	0.1554	0.6259
$\alpha = 2$	$\bar{T}$	2.1488	5.4478	11.4136	24.2979
$\beta = \text{OFF}$	σ_T	0.0491	0.1079	0.2354	0.5938
$\alpha = 5/2$	$\bar{T}$	2.2146	5.6488	11.6705	24.6764
$\beta = \text{OFF}$	σ_T	0.0479	0.1959	0.3204	0.5461

ตารางที่ 4.12 ค่า $\bar{T}$ และ σ_T ของ RAL-MSPSort ชนิดข้อมูล Uint32 และ Double ที่ $\theta = 1.00$ ในแต่ละค่า N , α และ β บนเครื่อง R9-2920 จำนวน 100 ครั้ง

α และ β	KPI	20M	50M	100M	200M
Uint32					
$\alpha = 5/3$	$\bar{T}$	1.1603	2.9786	6.2968	12.7600
$\beta = \text{OFF}$	σ_T	0.0167	0.0388	0.1980	0.5081
$\alpha = 2$	$\bar{T}$	1.1598	2.9678	6.2311	12.5458
$\beta = \text{OFF}$	σ_T	0.0146	0.0245	0.1192	0.2564
$\alpha = 5/2$	$\bar{T}$	1.1740	3.0219	6.4761	13.0133
$\beta = \text{OFF}$	σ_T	0.0245	0.0869	0.3986	0.7027
Double					
$\alpha = 3/2$	$\bar{T}$	1.7356	4.6483	9.6046	20.1066
$\beta = \text{OFF}$	σ_T	0.0557	0.1152	0.2038	0.5079
$\alpha = 5/3$	$\bar{T}$	1.7255	4.6356	9.4302	19.8885
$\beta = \text{OFF}$	σ_T	0.0725	0.1871	0.5151	0.9203
$\alpha = 2$	$\bar{T}$	1.7118	4.6077	9.5261	20.0296
$\beta = \text{OFF}$	σ_T	0.0795	0.2034	0.4592	1.0444
$\alpha = 5/2$	$\bar{T}$	1.7889	4.8657	10.2009	21.4310
$\beta = \text{OFF}$	σ_T	0.0724	0.3059	0.6525	1.7900

4.8.6 RAL-MSPSort และค่าทวิคูณเรต β

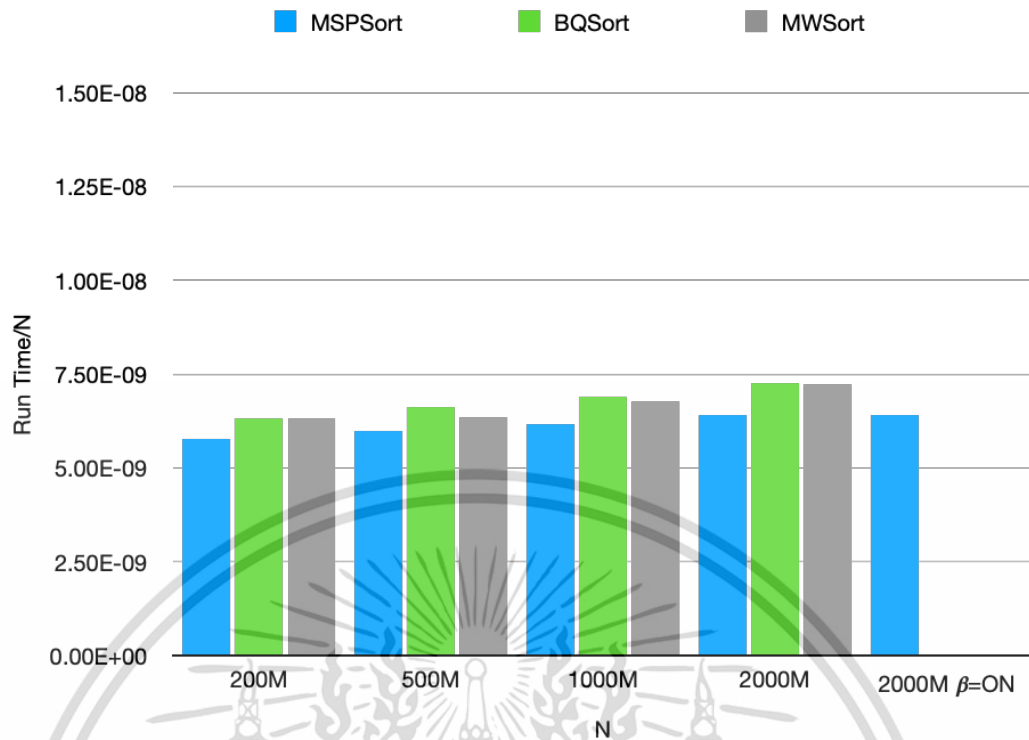
พารามิเตอร์สุดท้ายที่จะใช้ในการทดลองนี้คือค่า β ซึ่งจะมีการทดลองบน RAL-MSPSort จำนวน 1000 ครั้งเพื่อหาค่าที่เสถียรมากที่สุด และนำไปเปรียบเทียบกับ BQSort และ MWSort ผลลัพธ์ที่ได้ของเวลาที่ใช้จัดเรียงข้อมูลบน Pi4 และ R9-2920 จะนำไปแสดงบนตารางที่ ข.1 และ ข. 2 โดยผลลัพธ์ของเครื่อง Pi4 ที่ $N \leq 100M$ ส่วนใหญ่จะมีค่าใกล้เคียงกัน ในขณะที่ $N = 200M$ ค่า T_{O3} ถึง T_{Q1} เป็นค่าที่มีความเสถียรมากที่สุด บนเครื่อง R9-2920 RAL-MSPSort ใช้เวลาในการทำงานดีที่สุดบนชนิดข้อมูล Uint32 แต่ MWSort ใช้เวลาในการทำงานได้ดีที่สุดบนชนิดข้อมูล Double ดังภาพที่ 4.1 และ 4.2

ในการคำนวณหาค่า σ_T และ T_{QOR} พบว่าค่าทั้ง 2 ค่าของ RAL-MSPSort มีค่าน้อยกว่าทั้ง BQSort และ MWSort นั้นสรุปได้ว่า RAL-MSPSort มีความเสถียรมากกว่าทั้งเครื่องที่มีประสิทธิภาพต่ำและเครื่องที่มีประสิทธิภาพสูง

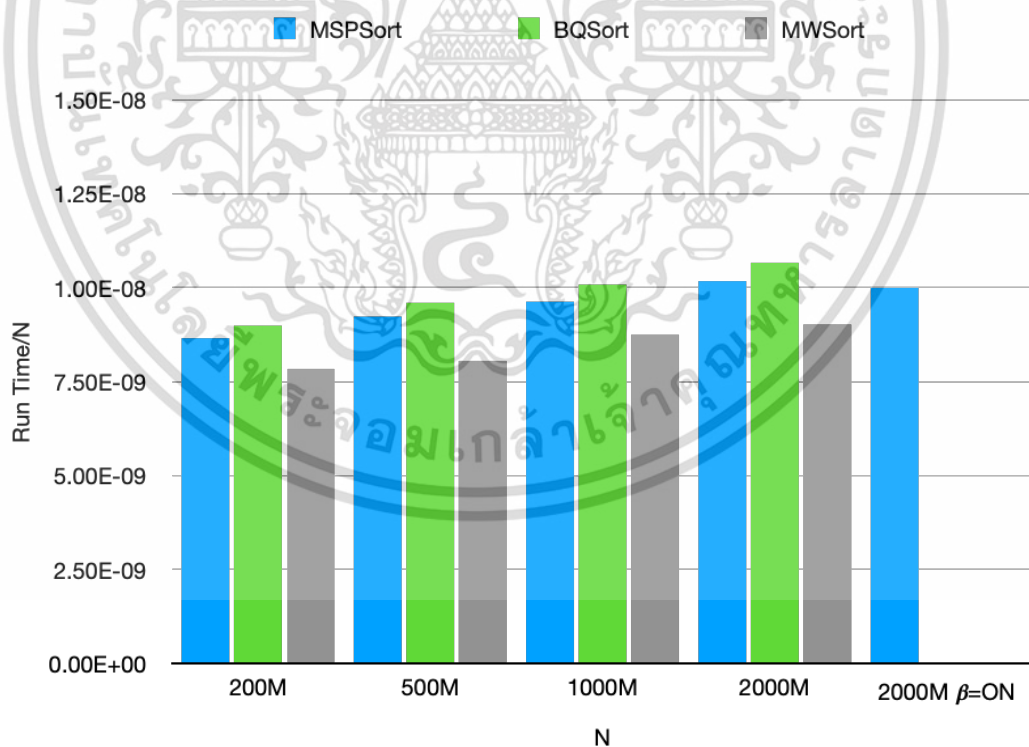
ในแง่ของการใช้งานซีพียู V_{usr} พบว่า RAL-MSPSort มีค่าร้อยละที่ต่ำกว่า BQSort แต่มีค่าสูงกว่า MWSort ซึ่งนั่นแปลว่าการทำงานเปรียบเทียบและสลับข้อมูล แบบ Hoare's ใน RAL-MSPSort และ BQSort ทำได้ในลักษณะเดียวกัน ต่างกันที่ BQSort ที่ใช้เทคนิคการขโมยงาน เพื่อให้ปริมาณงานของซีพียูมีสูงมากขึ้น RAL-MSPSort ใช้การจัดงานเรดของโอเพนเอ็มพี เพื่อให้การใช้งานซีพียูมีความใกล้เคียงกัน ดังนั้นอาจมีบางแกนของซีพียูที่ว่างลงและสามารถทำงานอื่นระหว่างที่จัดเรียงข้อมูลแบบขนาน

เมื่อมีงานที่ขนาดใหญ่มากขึ้น การใช้งานหน่วยความจำหลักต้องสูงมากขึ้นไปด้วย การใช้งานที่มากกว่า 100% ของขนาดข้อมูลจริงนั้นเป็นสิ่งที่ต้องนำมาวิเคราะห์เนื่องจากสิ่งนี้เป็นสิ่งที่เกิดขึ้นเนื่องจากการสร้างโครงสร้างข้อมูลเพื่อให้สามารถทำงานแบบขนานด้วยอัลกอริธึมที่แตกต่างกันไป ในการทดลองครั้งนี้ใช้ `/usr/bin/time` ในการวัดค่าพื้นที่หน่วยความจำที่ใช้ในการทำงานของอัลกอริธึม ซึ่งอัลกอริธึม RAL-MSPSort และ BQSort ใช้หน่วยความจำหลักในปริมาณที่ใกล้เคียงกันทั้งชนิดข้อมูล Uint32 และ Double ในทางกลับกัน MWSort ต้องใช้พื้นที่หน่วยความจำหลักเพิ่มขึ้น 2 เท่า ซึ่งเกิดจากฟังก์ชัน MultiWay Parallel Merge ซึ่งจะต้องใช้พื้นที่ของหน่วยความจำเพิ่มขึ้นส่งผลให้ $V_{ram} > 200\%$ ดังตารางที่ 4.13 และ 4.14 นั่นคือ MWSort ใช้ประโยชน์จากหน่วยความจำมากกว่าอัลกอริธึมเช่น RAL-MSPSort และ BQSort ที่มีรูปแบบอัลกอริธึมแบบ Hoare's

ค่า β นั้นมีการเปิดใช้งานบน $N=200M$ และ $2000M$ บน Pi4 และ R9-2920 ตามลำดับ ซึ่งผลลัพธ์ที่ได้แสดงในตารางที่ ข.3 และ ข.4 ด้านขวาสุด โดยค่า β_L และ β_R ตามสมการที่ 4.4 และ 4.5 เป็นการทำให้การเลือกไพวอทสมดุลมากขึ้น เนื่องจากอาเรย์ที่มีขนาดใหญ่จะสามารถทำงานได้พร้อมกันหลายเรด นอกจากนี้ยังเห็นได้ว่าที่ $N=200M$ บน Pi4 นั้นค่า $\bar{T}$ และ σ_T มีค่าที่แย่งเนื่องจากจำนวนแกนของ Pi4 มีเพียง 4 แกน อย่างไรก็ตาม R9-2920 แสดงให้เห็นว่า เมื่อตั้งค่า $\beta=ON$ ทำให้ค่าของเวลาที่ได้น้อยกว่าการตั้งค่า $\beta=OFF$ ทั้งชนิดตัวข้อมูล Uint32 และ Double ที่ $N=2000M$



ภาพที่ 4.1 $\bar{T}/N$ ของ RAL-MSPSort ชนิดข้อมูล Uint32 บนเครื่อง R9-2920 ที่ $B=128kB$
 $U_{sort}=8MB$, $\alpha=2$ ในแต่ละค่า N จำนวน 1000 ครั้ง



ภาพที่ 4.2 $\bar{T}/N$ ของ RAL-MSPSort ชนิดข้อมูล Double บนเครื่อง R9-2920 ที่ $B=128kB$
 $U_{sort}=8MB$, $\alpha=2$ ในแต่ละค่า N จำนวน 1000 ครั้ง

4.9 การเปรียบเทียบเวลาต่อข้อมูล 1 ล้านตัวต่อแกนประมวลผลกับอัลกอริธึมการจัดเรียงข้อมูลอื่น

ในการเปรียบเทียบเวลาในการจัดเรียงข้อมูลแบบขนานนั้นจำเป็นต้องคำนึงถึงจำนวนข้อมูลที่ใช้ และจำนวนแกนประมวลผล ที่นำมาเปรียบเทียบ เพื่อให้เกิดความเหมาะสมและทัดเทียมกันในทุกอัลกอริธึมที่จะใช้ในการเปรียบเทียบข้อมูล โดยตารางที่ 4.13 แสดงเวลาในการจัดเรียงข้อมูลแบบขนานต่อล้านข้อมูลต่อแกนประมวลผล เมื่อเปรียบเทียบกับอัลกอริธึมอื่น โดยมีการแสดงชนิดข้อมูล จำนวนข้อมูล และจำนวนแกนประมวลผลในงานวิจัยนั้น

จากตารางที่ 4.13 พบว่าเวลาในการจัดเรียงข้อมูลแบบขนานต่อล้านข้อมูลต่อแกนประมวลผลของ MSPSort ด้วยชนิดข้อมูล Uint32 มีค่าต่ำที่สุด ซึ่งบ่งบอกว่าอัลกอริธึม MSPSort ใช้เวลาในการจัดเรียงข้อมูลน้อยที่สุดเมื่อเปรียบเทียบกับอัลกอริธึมอื่น อย่างไรก็ตาม หากข้อมูล Uint64 อัลกอริธึม MWSort ใช้เวลาในการทำงานน้อยที่สุดเมื่อเทียบกับอัลกอริธึมอื่น แต่อย่างไรก็ตามหากเปรียบเทียบกับอัลกอริธึมที่มีการทำงานพื้นฐานเป็นควิกซอร์ต MSPSort จะใช้เวลาในการทำงานน้อยที่สุด

ตารางที่ 4.13 เวลาในการจัดเรียงข้อมูลแบบขนานต่อล้านข้อมูลต่อแกนประมวลผลเมื่อเปรียบเทียบกับอัลกอริธึมอื่น

อัลกอริธึม (ชนิดข้อมูล, N , c)	$\bar{T}$ (วินาที)	$\bar{T}/M$	$\bar{T}/M/c$
MSPSort (Uint32,200M,24)	1.18	5.91×10^{-9}	2.46×10^{-10}
MSPSort (Uint64,200M,24)	1.69	8.47×10^{-9}	3.53×10^{-10}
PQuickSort (Uint32,64M,32)[4]	5.56	5.56×10^{-6}	1.74×10^{-7}
PSRS (Uint32,64M,32)[4]	8.02	8.02×10^{-6}	2.50×10^{-7}
sort_omp_2.0 (Uint32,100M,4)[5]	7.50	7.50×10^{-8}	9.38×10^{-9}
BQSort (Uint32,200M,24)[6,7]	1.26	6.32×10^{-9}	2.63×10^{-10}
BQSort (Uint64,200M,24)[6,7]	1.75	8.75×10^{-9}	3.65×10^{-10}
MWSort (Uint32,200M,24)[6,7]	1.26	6.30×10^{-8}	2.63×10^{-10}
MWSort (Uint64,200M,24)[6,7]	1.47	7.35×10^{-8}	3.06×10^{-10}
psort2 (Uint64,100M,24)[11,12]	2.71	2.71×10^{-8}	1.13×10^{-9}
PPMQSort (Uint32,200M,16)[15]	5.89	2.95×10^{-8}	3.68×10^{-9}
PPMQSort (Uint64,200M,16)[15]	7.09	3.54×10^{-8}	4.43×10^{-9}
DPPSort _{TL} (Uint32,200M,8)[17]	3.34	1.67×10^{-8}	2.09×10^{-9}
DPPSort _{qsort} (Uint32,200M,8)[17]	5.47	2.73×10^{-8}	3.42×10^{-9}

บทที่ 5

บทสรุป

ในบทนี้จะนำเสนอสรุปผลการวิจัย ข้อเสนอแนะ และแนวทางการพัฒนาต่อของการจัดเรียงข้อมูลแบบขนานโดยใช้หลายสแต็ก

5.1 สรุปผลการวิจัย

วิทยานิพนธ์เรื่องนี้ได้มีการศึกษาอัลกอริธึมการจัดเรียงข้อมูลแบบขนานที่มีการประมวลผลบนซีพียูหลายแกนโดยใช้สแต็ก ซึ่งการจัดเรียงข้อมูลในลักษณะนี้มีหลายรูปแบบไม่ว่าจะเป็นการแบ่งข้อมูลแบบบล็อก [3,4,6,7] การแบ่งข้อมูลออกเป็นขนาดใหญ่มาก่อนเพื่อใช้ในการแบ่งข้อมูล [10,15,17] การแบ่งข้อมูลแบบหลายไพวอท [11,12,14] นอกจากนี้ยังมีอีกหลายอัลกอริธึมที่มีการแก้ปัญหาการจัดเรียงข้อมูลแบบขนานภายหลังจากการแบ่งข้อมูลแบบขนานขั้นต้นแล้ว [8,9] ซึ่งอัลกอริธึมการแบ่งข้อมูลเหล่านี้สามารถนำมาประยุกต์ใช้ในการพัฒนาอัลกอริธึมการจัดเรียงข้อมูลแบบขนานให้มีประสิทธิภาพบนซีพียูหลายแกนได้

การจัดเรียงข้อมูลแบบขนานโดยใช้หลายสแต็กที่ได้นำเสนอแบ่งออกเป็น MSPartition ซึ่งมีวิธีการแบ่งข้อมูลแบบบล็อก โดยใช้ 1 ไพวอท โดยจะมีการเปรียบเทียบและสลับข้อมูลจากบล็อกด้านซ้ายและบล็อกขวาเข้ามาถึงแนวแบ่ง แต่ละเรคอร์ดจะมีสแต็กซ้ายและด้านขวาเพื่อเก็บขอบเขตของบล็อกข้อมูลที่ต้องการแบ่งข้อมูล โดยจะแบ่งข้อมูลจนกระทั่งมีสแต็กข้างใดข้างหนึ่งว่าง ทำรีเคอร์ซีฟเพื่อคลีนอัปแล้วจึงแบ่งข้อมูลด้วยการแบ่งข้อมูลแบบ Lomuto's ซึ่งการแบ่งข้อมูลนั้นสามารถนำเอาไลบรารีโอเพนเอ็มพีมาใช้อัลกอริธึมมีการในการทำงานแบบขนาน

งานวิจัยในช่วงแรกก่อนการออปติไมซ์จะมีการทดสอบบนเครื่องที่มีหลายแกนรวมถึงเครื่องที่มีระบบการทำงานแบบ NUMA เปรียบเทียบกับ BQSort MWSort โดย MSPSort ใช้เวลาในการทำงานน้อยกว่า BQSort ทุกชนิดข้อมูลและขนาดข้อมูล ยกเว้น Intel Xeon X5670 และ R9-2920 อย่างไรก็ตาม MWSort สามารถทำงานได้ดีกว่าบน NUMA ทั้งข้อมูลชนิด Uint64 และ Double เมื่อเปรียบเทียบกับ MSPSort

การออปติไมซ์การจัดเรียงข้อมูลแบบขนานสามารถทำได้โดยปรับค่าพารามิเตอร์ต่างๆ เช่น ขนาดบล็อก การแก้ไขปัญหาวายวอร์ในการแบ่งข้อมูล การปรับค่าอัตราส่วนเรคอร์ดในการแบ่งข้อมูล การจัดวางเรคอร์ดแบบพลวัต การใช้อัลกอริธึม Depth First Traversal การปรับค่าเรคอร์ดต่ำสุด เมื่อมีการออปติไมซ์การจัดเรียงข้อมูลแบบขนานโดยใช้หลายสแต็ก จะพบว่าอัลกอริธึม RAL-MSPSort สามารถทำงานได้ดีเมื่อเปรียบเทียบกับ BQSort และ MWSort ในการทดลองนี้มีการเปรียบเทียบบนเครื่องที่มีขนาดเล็กโดยใช้ Raspberry Pi4 ซึ่งมีหน่วยประมวลผล ARM Cortex A72

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

ที่มี 4 แกน และเครื่องที่มีขนาดใหญ่ขึ้นคือ AMD ThreadRipper ซึ่งมี 24 แกน โดยมีการเปรียบเทียบกันในเรื่องของเวลาที่ใช้ในการทำงาน ประสิทธิภาพการใช้ซีพียู และหน่วยความจำหลัก โดยอัลกอริธึม RAL-MSPSort บนเครื่องขนาดเล็ก ขนาดใหญ่ ใช้เวลาในการจัดเรียงข้อมูลน้อยกว่า BQSort และ MWSort มีความเสถียรของเวลาที่ใช้ในการจัดเรียงข้อมูลสูงกว่า สามารถถึงประสิทธิภาพการใช้งานซีพียูได้มากกว่า BQSort และ MWSort ใช้พื้นที่หน่วยความจำหลักระหว่างการจัดเรียงข้อมูลใกล้เคียงกับ BQSort และใช้พื้นที่หน่วยความจำน้อยกว่า MWSort 2 เท่า ในทำนองเดียวกัน RAL-MSPSort บนเครื่องขนาดใหญ่ มีความเสถียรของเวลาในการจัดเรียงข้อมูลสูงกว่าทั้ง BQSort และมีความเสถียรของเวลาในการจัดเรียงข้อมูลใกล้เคียงกับ MWSort แต่ MSPSort มีประสิทธิภาพการใช้ซีพียูสูงกว่า ใช้หน่วยความจำหลักระหว่างการจัดเรียงข้อมูลใกล้เคียงกับ BQSort และใช้น้อยกว่า MWSort 2 เท่า

5.2 ข้อเสนอแนะ

ในการพัฒนาอัลกอริธึมการจัดเรียงข้อมูลแบบขนานนั้น ควรจะมีการพัฒนาการแบ่งข้อมูลในส่วนที่เป็นการแบ่งข้อมูล ซึ่งมีอยู่หลากหลายรูปแบบไม่ว่าจะเป็น 1 ไพอท หรือหลายไพอท โดยส่วนที่มีการทำงานแบบขนานโดยปกติแล้วจะต้องเป็นบริเวณที่ไม่เกี่ยวข้องกัน หากมีบริเวณที่เกี่ยวข้องกันจะต้องมีการทำซิงโครไนเซชัน ซึ่งทำให้เวลาในการทำงานมากขึ้น เนื่องจากแต่ละเธรดจะต้องมีการรอการทำงานกัน จึงทำให้เกิดอัลกอริธึมที่แก้ปัญหาเหล่านี้ เช่น การขโมยงาน อย่างไรก็ตาม การแบ่งข้อมูลแบบขนานมักจะมีการแบ่งงานในตอนต้นหลักๆ อยู่ 2 แบบคือ การแบ่งเป็นบล็อกเล็ก และการตัดแบ่งข้อมูลเป็นขนาดใหญ่ ซึ่งต่างมีข้อดีข้อเสียแตกต่างกัน การแบ่งเป็นบล็อกเล็กจะต้องมีการรวมข้อมูลมากในภายหลัง แต่การตัดแบ่งข้อมูลเป็นขนาดใหญ่จะมีเธรดที่ทำงานในช่วงต้นน้อย ซึ่งการแบ่งข้อมูลครั้งแรกมักจะใช้เวลาในการทำงานมากที่สุด จึงจะต้องมีการจัดการเธรดที่เหมาะสม นอกจากนี้การทำงานแบบขนานจะมีโอเวอร์เฮด หากนักพัฒนาเลือกบริเวณที่ทำงานแบบขนานที่มีปริมาณงานน้อย จะทำให้โอเวอร์เฮดการทำงานสูงกว่าเวลาที่ใช้ทำงานจริง จึงไม่เกิดความคุ้มค่าในการใช้งานอัลกอริธึมแบบขนาน อย่างไรก็ตามขนาดของข้อมูลมีจำนวนเพิ่มมากขึ้นอย่างก้าวกระโดด ทำให้การพัฒนาอัลกอริธึมแบบขนานจึงมีความสำคัญเพิ่มมากขึ้นอย่างไม่สามารถหลีกเลี่ยงได้

จากงานภายในวิทยานิพนธ์ฉบับนี้จะเห็นว่าสามารถนำเอาอัลกอริธึมไปประยุกต์ใช้บนเครื่องหลากหลายรูปแบบไม่ว่าจะเป็นเครื่องแม่ข่าย เครื่องเวิร์คสเตชัน หรือแม้กระทั่งอุปกรณ์พกพา เช่น สมาร์ทโฟน ซึ่งจากการออกแบบไมซ์จะเห็นว่าบนเครื่องขนาดเล็กไม่ว่าจะเป็นสมาร์ทโฟน แท็บเล็ต หรืออุปกรณ์พกพาอื่น พารามิเตอร์ที่เหมาะสมเรียงตามลำดับ B: U_{sort} : m: τ_{min} : α : β คือ 128kB: 8MB: 1: 2: 2: OFF ตามลำดับ ในทางกลับกันเครื่องที่มีขนาดใหญ่ไม่ว่าจะเป็นเครื่องเวิร์คสเตชัน พารามิเตอร์ที่เหมาะสมคือ 128kB: 8MB: 1: 2: 2: ON ตามลำดับ

ภายในวิทยานิพนธ์นี้มีการนำเอาอัลกอริธึมควิกซอร์ตมาพัฒนาให้มีความทำงานแบบขนาน ทำให้คุณลักษณะพื้นฐานในการจัดเรียงข้อมูลมีลักษณะเดียวกันกับอัลกอริธึมควิกซอร์ต ไม่ว่าจะเป็นลักษณะการกระจายตัวของข้อมูลที่เหมาะสมในการจัดเรียงข้อมูล หรือแม้กระทั่งการใช้พื้นที่ในการจัดเรียงข้อมูล โดยในงานนี้จะเห็นว่าการทดลองโดยใช้ชนิดข้อมูลแบบสุ่มที่มีการกระจายตัวแบบยูนิฟอร์ม ซึ่งเป็นการกระจายตัวพื้นฐานของข้อมูลทั่วไป อย่างไรก็ตาม การกระจายตัวที่เหมาะสมกับอัลกอริธึมประเภทควิกซอร์ตมีอีกหลายรูปแบบไม่ว่าจะเป็นข้อมูลเรียงแบบย้อนกลับ (Reversed) ข้อมูลที่กำลังจะจัดเรียงสำเร็จ (Nearly Sorted) ซึ่งล้วนแล้วแต่ใช้เวลาน้อยกว่าการกระจายตัวแบบสุ่มแบบยูนิฟอร์ม ทั้งสิ้น ซึ่งผลลัพธ์ของอัลกอริธึมควิกซอร์ตที่มีความทำงานแบบขนานแสดงไว้ในหลายงานวิจัย เช่น [13,15]

งานวิจัยภายในวิทยานิพนธ์เล่มนี้สามารถนำเอาไปใช้ในแอปพลิเคชันได้หลากหลายรูปแบบ ไม่ว่าจะเป็นการจัดเรียงข้อมูลภายในฐานข้อมูล อัลกอริธึมทางในการพยากรณ์อากาศ (Weather Prediction) อัลกอริธึมทางด้านไบโออินฟอร์เมติกส์ (Bioinformatics) ซึ่งอัลกอริธึมเหล่านี้ล้วนแล้วแต่มีปริมาณข้อมูลอินพุตที่มหาศาล มีการนำไปประมวลผล และมีการจัดเรียงข้อมูล ซึ่งต้องใช้เวลาในการประมวลผลเป็นอย่างมาก โดยหากไม่นำอัลกอริธึมที่ทำงานแบบขนานมาประมวลผลจะทำให้สามารถใช้ทรัพยากรหน่วยประมวลผล ได้เพียง 1 แกนประมวลผล ส่งผลให้มีการนำเอาทรัพยากรในการประมวลผลมาใช้ไม่มีประสิทธิภาพ

5.3 แนวทางการพัฒนาต่อ

แนวทางในการพัฒนาต่อของวิทยานิพนธ์นี้สามารถปรับการทำงานให้ทำงานแบบหลายโพรเซสเซอร์ การเปลี่ยนโครงสร้างข้อมูลในการแบ่งข้อมูลจากสแต็กที่มีลักษณะเป็นคู่ ให้กลายเป็น deque ซึ่งในแต่ละเรคคอร์ดจะใช้ 1 deque เพื่อลดจำนวนขั้นตอนของการทำรีเคอซีฟ การปรับการจัดการงานโดยอาจจะนำเอาการจัดงานแบบ SPF ทำงานร่วมกันกับ RAL และออกแบบวิธีการทำงานของอัลกอริธึมที่นำเสนอ หรือการนำเอาอัลกอริธึมการแบ่งงานและจัดการงานไปทำงานบนหน่วยประมวลผลกราฟิก เพื่อให้การทำงานมีประสิทธิภาพมากขึ้น นอกจากนี้ยังสามารถทดลองอัลกอริธึมบนการกระจายตัวของข้อมูลในแบบต่างๆ การเลือกโพรเซสเซอร์โดยคำนึงถึงโพรเซสเซอร์ที่ใช้แล้วในการแบ่งข้อมูลครั้งก่อนหน้ามาประกอบ เพื่อให้โพรเซสเซอร์ที่ได้สามารถนำมาใช้ในการแบ่งข้อมูลได้ดีขึ้น

เอกสารอ้างอิง

- [1] Hoare, C.A.R., 1961. "Algorithm 63: partition." **Communications of the ACM**. 4(7) : 321.
- [2] Bentley, J.L. and McIlroy, M.D., 1993. "Engineering a sort function." **Software: Practice and Experience**. 23(11) : 1249-1265.
- [3] Heidelberger, P., Norton, A. and Robinson, J.T., 1990. "Parallel quicksort using fetch-and-add." **IEEE Transactions on Computers**, 39(1) : 133-138.
- [4] Tsigas, P. and Zhang, Y., 2003. "A simple, fast parallel implementation of quicksort and its performance evaluation on SUN enterprise 10000." In **Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing** : 372-381.
- [5] Süß, M. and Leopold, C., 2004, October. "A user's experience with parallel sorting and OpenMP." In **Proceedings of the Sixth European Workshop on OpenMP-EWOMP'04** : 23-38.
- [6] Singler, J., Sanders, P. and Putze, F., 2007. "MCSTL: The multi-core standard template library." In **European Conference on Parallel Processing** : 682-694.
- [7] Putze, F., Sanders, P. and Singler, J., 2007. "MCSTL: the multi-core standard template library." In **Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming** : 144-145.
- [8] Frias, L. and Petit, J., 2008. "Parallel partition revisited." In **International Workshop on Experimental and Efficient Algorithms** : 142-153.
- [9] Traoré, D., Roch, J.L., Maillard, N., Gautier, T. and Bernard, J., 2008. "Deque-free work-optimal parallel STL algorithms." In **European Conference on Parallel Processing** : 887-897.
- [10] Ayguadé, E., Copt, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P. and Zhang, G., 2008. "The design of OpenMP tasks." **IEEE Transactions on Parallel and Distributed Systems**, 20(3) : 404-418.

- [11] Nakagawa, M., Man, D., Ito, Y. and Nakano, K., 2009. "A simple parallel convex hulls algorithm for sorted points and the performance evaluation on the multicore processors." In **2009 International conference on parallel and distributed computing, applications and technologies** : 506-511.
- [12] Man, D., Ito, Y. and Nakano, K., 2011. "An efficient parallel sorting compatible with the standard qsort." **International Journal of Foundations of Computer Science**, 22(05) : 1057-1071.
- [13] Cederman, D. and Tsigas, P., 2010. "Gpu-quicksort: A practical quicksort algorithm for graphics processors." **Journal of Experimental Algorithmics (JEA)**, 14 : 1-4.
- [14] Mahafzah, B.A., 2013. "Performance assessment of multithreaded quicksort algorithm on simultaneous multithreaded architecture." **The Journal of Supercomputing**, 66(1) : 339-363.
- [15] Ranokphanuwat, R. and Kittitornkun, S., 2016. "Parallel partition and merge QuickSort (PPMQSort) on multicore CPUs." **The Journal of Supercomputing**, 72(3) : 1063-1091.
- [16] Axtmann, M., Witt, S., Ferizovic, D. and Sanders, P., 2017. "In-place parallel super scalar samplesort (ipssso)". In **25th Annual European Symposium on Algorithms (ESA 2017)**. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
- [17] Rattanatanurak, A., 2018. "Dual Parallel Partition Sorting Algorithm." In **Proceedings of 2018 the 8th International Workshop on Computer Science and Engineering** : 685-690.
- [18] Singler, J. and Konsik, B., 2008. "The GNU libstdc++ parallel mode: software engineering considerations." In **Proceedings of the 1st international workshop on Multicore software engineering** : 15-22.
- [19] "OpenMP Application Programming Interface version 5.0" [Online]. Available: <http://openmp.org>, 2018.

ภาคผนวก ก

ก.1 ผลลัพธ์ของเวลาที่ใช้ในการจัดเรียงข้อมูล 1000 ครั้ง

ตารางที่ ก.1 เวลาที่ใช้ในการจัดเรียง $\bar{T}$ ของ MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิดข้อมูล Uint32 ตามขนาด N บนเครื่อง i7-2600 โดยทดลอง 1000 ครั้ง

อัลกอริธึม	ตัวชี้วัด (วินาที)	200M	500M	1000M	2000M
MPSort	T_{Q1}	3.042	8.113	17.832	37.880
	T_{Q2}	3.073	8.179	17.963	38.340
	$\bar{T}$	3.182	8.342	17.928	38.332
	T_{Q3}	3.318	8.283	18.039	38.797
	σ_T	0.196	0.436	0.171	0.536
BQSort	T_{Q1}	3.212	8.578	18.285	38.670
	T_{Q2}	3.247	8.665	18.447	39.105
	$\bar{T}$	3.348	8.856	18.484	39.503
	T_{Q3}	3.491	8.804	18.663	40.130
	σ_T	0.198	0.503	0.270	1.111
MWSort	T_{Q1}	3.649	9.550	20.132	40.920
	T_{Q2}	3.700	9.675	20.382	41.588
	$\bar{T}$	3.812	9.710	20.385	41.764
	T_{Q3}	4.016	9.812	20.633	42.470
	σ_T	0.231	0.266	0.383	1.105

ตารางที่ ก.2 เวลาที่ใช้ในการจัดเรียง $\bar{T}$ ของ MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิดข้อมูล Uint64 ตามขนาด N บนเครื่อง i7-2600 โดยทดลอง 1000 ครั้ง

อัลกอริธึม	ตัวชี้วัด (วินาที)	200M	500M	1000M	2000M
MSPSort	T_{Q1}	3.648	9.855	21.540	44.592
	T_{Q2}	3.772	9.909	21.725	44.813
	$\bar{T}$	3.813	9.956	21.712	44.887
	T_{Q3}	4.065	9.973	21.893	45.104
	σ_T	0.205	0.234	0.243	0.454
BQSort	T_{Q1}	3.702	10.023	21.780	45.898
	T_{Q2}	3.767	10.104	21.976	46.332
	$\bar{T}$	3.877	10.254	21.977	46.511
	T_{Q3}	4.151	10.233	22.144	47.073
	σ_T	0.227	0.442	0.270	0.776
MWSort	T_{Q1}	4.194	11.202	23.721	49.292
	T_{Q2}	4.253	11.312	23.947	49.633
	$\bar{T}$	4.326	11.338	23.975	49.703
	T_{Q3}	4.360	11.449	24.218	50.044
	σ_T	0.209	0.233	0.391	0.709

ตารางที่ ก.3 เวลาที่ใช้ในการจัดเรียง $\bar{T}$ ของ MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิดข้อมูล Double ตามขนาด N บนเครื่อง i7-2600 โดยทดลอง 1000 ครั้ง

อัลกอริธึม	ตัวชี้วัด (วินาที)	200M	500M	1000M	2000M
MSPSort	T_{Q1}	3.851	10.521	22.908	48.058
	T_{Q2}	3.917	10.595	23.048	48.684
	$\bar{T}$	4.013	10.725	23.050	49.038
	T_{Q3}	4.110	10.693	23.187	50.166
	σ_T	0.222	0.422	0.202	1.151
BQSort	T_{Q1}	3.937	10.754	23.399	49.533
	T_{Q2}	4.093	10.962	23.711	50.771
	$\bar{T}$	4.197	11.235	23.769	50.883
	T_{Q3}	4.413	11.283	24.183	52.384
	σ_T	0.266	0.706	0.458	1.484
MWSort	T_{Q1}	4.247	11.361	24.243	50.250
	T_{Q2}	4.522	11.873	25.080	51.807
	$\bar{T}$	4.522	11.857	25.122	52.190
	T_{Q3}	4.696	12.213	25.966	54.108
	σ_T	0.312	0.607	0.939	2.212

ตารางที่ ก.4 เวลาที่ในการจัดเรียง $\bar{T}$ ของ MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิดข้อมูล Uint32 ตามขนาด N บนเครื่อง R7-1700 โดยทดลอง 1000 ครั้ง

อัลกอริธึม	ตัวชี้วัด (วินาที)	200M	500M	1000M	2000M
MSPSort	T_{Q1}	1.722	4.416	9.382	19.238
	T_{Q2}	1.735	4.438	9.413	19.294
	$\bar{T}$	1.746	4.476	9.418	19.307
	T_{Q3}	1.773	4.561	9.445	19.353
	σ_T	0.032	0.083	0.063	0.120
BQSort	T_{Q1}	1.780	4.643	9.897	20.358
	T_{Q2}	1.811	4.722	10.051	20.695
	$\bar{T}$	1.807	4.725	10.026	20.635
	T_{Q3}	1.827	4.778	10.125	20.845
	σ_T	0.032	0.101	0.160	0.316
MWSort	T_{Q1}	1.973	5.096	10.436	21.290
	T_{Q2}	2.145	5.470	11.114	22.498
	$\bar{T}$	2.109	5.389	10.959	22.214
	T_{Q3}	2.187	5.549	11.241	22.696
	σ_T	0.041	0.086	0.146	0.236

ตารางที่ ก.5 เวลาที่ในการจัดเรียง $\bar{T}$ ของ MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิดข้อมูล Uint64 ตามขนาด N บนเครื่อง R7-1700 โดยทดลอง 1000 ครั้ง

อัลกอริธึม	ตัวชี้วัด (วินาที)	200M	500M	1000M	2000M
MSPSort	T_{Q1}	2.149	5.723	12.046	25.406
	T_{Q2}	2.161	5.744	12.091	25.494
	$\bar{T}$	2.163	5.752	12.104	25.514
	T_{Q3}	2.174	5.772	12.144	25.584
	σ_T	0.022	0.048	0.092	0.168
BQSort	T_{Q1}	2.137	5.706	11.990	25.231
	T_{Q2}	2.153	5.746	12.077	25.423
	$\bar{T}$	2.160	5.762	12.102	25.487
	T_{Q3}	2.177	5.805	12.196	25.671
	σ_T	0.033	0.083	0.158	0.348
MWSort	T_{Q1}	2.216	5.845	12.184	25.469
	T_{Q2}	2.225	5.864	12.223	25.625
	$\bar{T}$	2.227	5.868	12.228	26.223
	T_{Q3}	2.236	5.887	12.268	27.148
	σ_T	0.015	0.033	0.063	0.890

ตารางที่ ก.6 เวลาที่ใช้ในการจัดเรียง $\bar{T}$ ของ MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิดข้อมูล Double ตามขนาด N บนเครื่อง R7-1700 โดยทดลอง 1000 ครั้ง

อัลกอริธึม	ตัวชี้วัด (วินาที)	200M	500M	1000M	2000M
MSPSort	T_{Q1}	2.312	6.094	12.699	26.616
	T_{Q2}	2.324	6.120	12.749	26.720
	$\bar{T}$	2.327	6.125	12.757	26.745
	T_{Q3}	2.338	6.146	12.805	26.829
	σ_T	0.026	0.046	0.090	0.210
BQSort	T_{Q1}	2.312	6.097	12.721	26.568
	T_{Q2}	2.327	6.134	12.799	26.751
	$\bar{T}$	2.333	6.147	12.830	26.810
	T_{Q3}	2.347	6.188	12.906	26.942
	σ_T	0.030	0.074	0.159	0.631
MWSort	T_{Q1}	2.735	7.037	14.366	29.394
	T_{Q2}	2.774	7.106	14.485	29.608
	$\bar{T}$	2.778	7.121	14.505	29.628
	T_{Q3}	2.818	7.196	14.626	29.838
	σ_T	0.051	0.120	0.191	0.333

ตารางที่ ก.7 เวลาที่ใช้ในการจัดเรียง $\bar{T}$ ของ MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิดข้อมูล Uint32 ตามขนาด N บนเครื่อง X5670 โดยทดลอง 1000 ครั้ง

อัลกอริธึม	ตัวชี้วัด (วินาที)	200M	500M	1000M	2000M
MSPSort	T_{Q1}	1.587	4.139	8.334	16.708
	T_{Q2}	1.601	4.177	8.408	16.845
	$\bar{T}$	1.605	4.184	4.184	16.907
	T_{Q3}	1.618	4.216	4.216	17.000
	σ_T	0.027	0.072	0.072	0.334
BQSort	T_{Q1}	1.684	4.039	8.145	16.576
	T_{Q2}	1.692	4.057	8.176	16.662
	$\bar{T}$	1.691	4.073	8.235	16.757
	T_{Q3}	1.699	4.088	8.223	16.788
	σ_T	0.011	0.063	0.168	0.304
MWSort	T_{Q1}	1.686	4.039	8.155	16.642
	T_{Q2}	1.693	4.055	8.183	16.708
	$\bar{T}$	1.692	4.070	8.235	16.819
	T_{Q3}	1.699	4.078	8.223	16.840
	σ_T	0.011	0.062	0.168	0.300

ตารางที่ ก.8 เวลาที่ใช้ในการจัดเรียง $\bar{T}$ ของ MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิดข้อมูล Uint64 ตามขนาด N บนเครื่อง X5670 โดยทดลอง 1000 ครั้ง

อัลกอริธึม	ตัวชี้วัด (วินาที)	200M	500M	1000M	2000M
MSPSort	T_{Q1}	2.696	6.694	13.360	24.210
	T_{Q2}	2.736	6.829	13.663	24.831
	$\bar{T}$	2.746	6.843	13.688	25.004
	T_{Q3}	2.788	6.980	14.024	25.582
	σ_T	0.072	0.206	0.466	1.065
BQSort	T_{Q1}	2.543	6.340	12.695	24.110
	T_{Q2}	2.584	6.344	12.951	24.953
	$\bar{T}$	2.601	6.417	13.021	25.315
	T_{Q3}	2.638	6.525	13.270	26.085
	σ_T	0.086	0.275	0.491	1.767
MWSort	T_{Q1}	2.065	5.103	10.166	NA
	T_{Q2}	2.085	5.133	10.753	NA
	$\bar{T}$	2.078	5.170	10.693	NA
	T_{Q3}	2.100	5.205	10.880	NA
	σ_T	0.035	0.113	0.492	NA

ตารางที่ ก.9 เวลาที่ใช้ในการจัดเรียง $\bar{T}$ ของ MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิดข้อมูล Double ตามขนาด N บนเครื่อง X5670 โดยทดลอง 1000 ครั้ง

อัลกอริธึม	ตัวชี้วัด (วินาที)	200M	500M	1000M	2000M
MSPSort	T_{Q1}	2.737	6.672	13.497	24.569
	T_{Q2}	2.771	6.774	13.808	25.186
	$\bar{T}$	2.780	6.796	13.835	25.334
	T_{Q3}	2.812	6.892	14.139	25.808
	σ_T	0.068	0.186	0.468	1.065
BQSort	T_{Q1}	2.610	6.414	13.032	24.890
	T_{Q2}	2.647	6.495	13.248	25.601
	$\bar{T}$	2.664	6.534	13.321	25.981
	T_{Q3}	2.697	6.603	13.547	26.725
	σ_T	0.078	0.187	0.441	1.650
MWSort	T_{Q1}	2.245	5.711	11.717	NA
	T_{Q2}	2.262	5.769	11.787	NA
	$\bar{T}$	2.258	5.724	11.790	NA
	T_{Q3}	2.277	5.811	11.878	NA
	σ_T	0.031	0.149	0.197	NA

ตารางที่ ก.10 เวลาที่ในการจัดเรียง $\bar{T}$ ของ MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิดข้อมูล Uint32 ตามขนาด N บนเครื่อง R9-2920 โดยทดลอง 1000 ครั้ง

อัลกอริธึม	ตัวชี้วัด (วินาที)	200M	500M	1000M	2000M
MSPSort	T_{Q1}	1.171	2.972	6.124	12.589
	T_{Q2}	1.181	2.991	6.157	12.659
	$\bar{T}$	1.182	2.994	6.173	12.681
	T_{Q3}	1.191	3.012	6.203	12.730
	σ_T	0.017	0.033	0.086	0.158
BQSort	T_{Q1}	1.237	3.221	6.727	13.971
	T_{Q2}	1.261	3.287	6.859	14.375
	$\bar{T}$	1.264	3.303	6.891	14.488
	T_{Q3}	1.285	3.352	6.991	14.831
	σ_T	0.040	0.127	0.288	0.774
MWSort	T_{Q1}	1.237	3.125	6.423	14.343
	T_{Q2}	1.248	3.142	6.850	14.524
	$\bar{T}$	1.260	3.175	6.774	14.454
	T_{Q3}	1.269	3.197	6.942	14.686
	σ_T	0.035	0.075	0.294	0.367

ตารางที่ ก.11 เวลาที่ในการจัดเรียง $\bar{T}$ ของ MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิดข้อมูล Uint64 ตามขนาด N บนเครื่อง R9-2920 โดยทดลอง 1000 ครั้ง

อัลกอริธึม	ตัวชี้วัด (วินาที)	200M	500M	1000M	2000M
MSPSort	T_{Q1}	1.680	4.514	9.602	20.180
	T_{Q2}	1.691	4.547	9.678	20.353
	$\bar{T}$	1.694	4.556	9.690	20.357
	T_{Q3}	1.703	4.588	9.771	20.537
	σ_T	0.023	0.065	0.148	0.330
BQSort	T_{Q1}	1.703	4.549	9.529	20.332
	T_{Q2}	1.732	4.638	9.742	20.815
	$\bar{T}$	1.746	4.682	9.838	20.980
	T_{Q3}	1.775	4.769	10.048	21.448
	σ_T	0.063	0.205	0.483	0.990
MWSort	T_{Q1}	1.457	3.898	8.106	16.388
	T_{Q2}	1.474	3.997	8.207	16.584
	$\bar{T}$	1.474	3.946	8.190	16.582
	T_{Q3}	1.489	4.050	8.300	16.766
	σ_T	0.027	0.159	0.187	0.356

ตารางที่ ก.12 เวลาที่ในการจัดเรียง $\bar{T}$ ของ MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิดข้อมูล Double ตามขนาด N บนเครื่อง R9-2920 โดยทดลอง 1000 ครั้ง

อัลกอริธึม	ตัวชี้วัด (วินาที)	200M	500M	1000M	2000M
MSPSort	T_{Q1}	1.747	4.679	9.757	20.611
	T_{Q2}	1.759	4.708	9.826	20.756
	$\bar{T}$	1.762	4.718	9.837	20.777
	T_{Q3}	1.772	4.744	9.905	20.906
	σ_T	0.024	0.069	0.118	0.263
BQSort	T_{Q1}	1.756	4.677	9.806	20.655
	T_{Q2}	1.782	4.760	10.003	21.051
	$\bar{T}$	1.798	4.799	10.081	21.306
	T_{Q3}	1.826	4.871	10.253	21.688
	σ_T	0.059	0.173	0.418	1.047
MWSort	T_{Q1}	1.554	3.938	8.791	17.919
	T_{Q2}	1.566	3.960	8.877	18.096
	$\bar{T}$	1.570	4.028	8.732	18.044
	T_{Q3}	1.582	4.002	8.936	18.243
	σ_T	0.024	0.160	0.342	0.406

ภาคผนวก ข

ข.1 ผลลัพธ์ของเวลาที่ใช้ในการจัดเรียงข้อมูล RAL-MSPSort ในการทดลองหาค่า β

ตารางที่ ข.1 ตัวชี้วัดของ RAL-MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิดข้อมูล Uint32 ที่ $\theta = 1.00$ ในแต่ละ N บนเครื่อง Pi4 จำนวน 1000 ครั้ง เมื่อ $B=128\text{KB}$, $U_{\text{sort}} = 8\text{MB}$, $m=1$, $\tau_{\min}=2$, $\alpha = 2$, $\beta=\text{OFF}$

อัลกอริธึม	ตัวชี้วัด	20M	50M	100M	200M	200M $\beta=\text{ON}$
RAL-MSPSort	T_{Q3}	1.2949	3.3129	6.9138	14.4948	14.6631
	$\bar{T}$	1.2749	3.2883	6.8673	14.4120	14.5390
	T_{Q2}	1.2732	3.2866	6.8607	14.3877	14.4783
	T_{Q1}	1.2539	3.2622	6.8136	14.3117	14.3709
	T_{IQR}	0.0410	0.0507	0.1002	0.1831	0.2922
	σ_T	0.0306	0.0374	0.0875	0.1626	0.2313
	$V_{\text{usr}}(\%)$	92.1907	95.8898	97.1730	97.7721	97.7858
	$V_{\text{ram}}(\%)$	103.7916	101.6674	100.8546	100.5402	
BQSort	T_{Q3}	1.2814	3.4008	7.1837	14.9355	
	$\bar{T}$	1.2682	3.3726	7.1146	14.8235	
	T_{Q2}	1.2640	3.3663	7.0259	14.8122	
	T_{Q1}	1.2482	3.3389	7.0907	14.6850	
	T_{IQR}	0.0332	0.0619	0.1578	0.2505	
	σ_T	0.0327	0.0480	0.1268	0.1986	
	$V_{\text{usr}}(\%)$	98.2782	98.6422	98.8069	98.9111	
	$V_{\text{ram}}(\%)$	103.9911	101.5996	100.7971	100.4020	
MWSort	T_{Q3}	1.2517	3.2947	6.9192	14.6352	
	$\bar{T}$	1.2401	3.2739	6.8760	14.5602	
	T_{Q2}	1.2354	3.2678	6.8708	14.5445	
	T_{Q1}	1.2243	3.2469	6.8253	14.4635	
	T_{IQR}	0.0274	0.0478	0.0939	0.1717	
	σ_T	0.0243	0.0400	0.0693	0.1453	
	$V_{\text{usr}}(\%)$	93.3338	93.8942	94.2563	94.5216	
	$V_{\text{ram}}(\%)$	203.8262	201.5361	200.7799	200.3928	

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

ตารางที่ ข.2 ตัวชี้วัดของ RAL-MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิดข้อมูล Double ที่ $\theta = 1.00$ ในแต่ละ N บนเครื่อง Pi4 จำนวน 1000 ครั้ง เมื่อ $B=128\text{KB}$, $U_{\text{sort}} = 8\text{MB}$, $m=1$, $\tau_{\min}=2$, $\alpha=2$, $\beta=\text{OFF}$

อัลกอริธึม	ตัวชี้วัด	20M	50M	100M	200M	200M $\beta=\text{ON}$
RAL-MSPSort	T_{Q3}	2.1275	5.5136	11.6239	24.6005	24.6358
	$\bar{T}$	2.0899	5.4371	11.4603	24.2545	24.3857
	T_{Q2}	2.0806	5.4507	11.4952	24.2736	24.3834
	T_{Q1}	2.0452	5.3733	11.2931	23.8399	24.1348
	T_{IQR}	0.0823	0.1403	0.3308	0.7606	0.5010
	σ_T	0.0610	0.1154	0.2571	0.5938	0.4926
	$V_{\text{usr}}(\%)$	93.3805	96.2759	97.2177	97.6680	97.7014
	$V_{\text{ram}}(\%)$	100.8292	100.7939	100.4250	100.2635	
BQSort	T_{Q3}	2.0836	5.5412	11.7007	24.8755	
	$\bar{T}$	2.0442	5.4775	11.5933	24.6479	
	T_{Q2}	2.0424	5.4627	11.5666	24.6200	
	T_{Q1}	1.9991	5.4014	11.4318	24.3610	
	T_{IQR}	0.0845	0.1398	0.2689	0.5145	
	σ_T	0.0540	0.1134	0.2541	0.4409	
	$V_{\text{usr}}(\%)$	97.9326	98.2525	98.4060	98.5310	
	$V_{\text{ram}}(\%)$	101.9699	100.7939	100.3972	100.1997	
MWSort	T_{Q3}	1.9794	5.3865	11.6665	24.9317	
	$\bar{T}$	1.9667	5.3554	11.5684	24.8053	
	T_{Q2}	1.9639	5.3343	11.5659	24.6787	
	T_{Q1}	1.9504	5.2840	11.4709	24.4965	
	T_{IQR}	0.0290	0.1025	0.1956	0.4352	
	σ_T	0.0249	0.0826	0.1848	0.5598	
	$V_{\text{usr}}(\%)$	90.9457	91.7825	92.3183	92.7510	
	$V_{\text{ram}}(\%)$	201.9203	200.7772	200.3857	200.1931	

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

ตารางที่ ข.3 ตัวชี้วัดของ RAL-MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิดข้อมูล Uint32 ที่ $\theta = 1.00$ ในแต่ละ N บนเครื่อง R9-2920 จำนวน 1000 ครั้ง เมื่อ $B=128KB$, $U_{sort} = 8MB$, $m=1$, $\tau_{min}=2$, $\alpha=2$, $\beta=OFF$

อัลกอริธึม	ตัวชี้วัด	20M	50M	100M	200M	200M $\beta=ON$
RAL-MSPSort	T_{Q3}	1.1615	3.0147	6.2208	12.8731	12.7956
	$\bar{T}$	1.1532	2.9871	6.1665	12.8096	12.6881
	T_{Q2}	1.1511	2.9720	6.1285	12.7004	12.6646
	T_{Q1}	1.1416	2.9378	6.0565	12.5842	12.5497
	T_{IQR}	0.0199	0.0765	0.1743	0.2889	0.2459
	σ_T	0.0194	0.0245	0.1192	0.2564	0.1976
	$V_{usr}(\%)$	90.7344	93.3419	94.0156	94.0447	95.3851
	$V_{ram}(\%)$	100.5634	100.3763	100.2624	100.2141	
BQSort	T_{Q3}	1.2849	3.3515	6.9910	14.8314	
	$\bar{T}$	1.2641	3.3027	6.8910	14.4880	
	T_{Q2}	1.2608	3.2872	6.8587	14.3754	
	T_{Q1}	1.2366	3.2211	6.7273	13.9707	
	T_{IQR}	0.0483	0.1304	0.2637	0.8607	
	σ_T	0.0398	0.1266	0.2883	0.7737	
	$V_{usr}(\%)$	94.2977	94.8018	94.3827	91.0273	
	$V_{ram}(\%)$	100.5719	100.2307	100.1184	100.0607	
MWSort	T_{Q3}	1.2690	3.1965	6.9421	14.6856	
	$\bar{T}$	1.2605	3.1748	6.7743	14.4544	
	T_{Q2}	1.2482	3.1424	6.8500	14.5239	
	T_{Q1}	1.2368	3.1253	6.4225	14.3433	
	T_{IQR}	0.0322	0.0712	0.5196	0.3423	
	σ_T	0.0351	0.0748	0.2940	0.3672	
	$V_{usr}(\%)$	89.3874	91.3076	88.7198	85.9254	
	$V_{ram}(\%)$	200.5459	200.2186	200.1084	200.0546	

ตารางที่ ข.4 ตัวชี้วัดของ RAL-MSPSort เปรียบเทียบกับ BQSort และ MWSort ชนิดข้อมูล Double ที่ $\theta = 1.00$ ในแต่ละ N บนเครื่อง R9-2920 จำนวน 1000 ครั้ง เมื่อ $B=128\text{KB}$, $U_{\text{sort}} = 8\text{MB}$, $m=1$, $\tau_{\min}=2$, $\alpha=2$, $\beta=\text{OFF}$

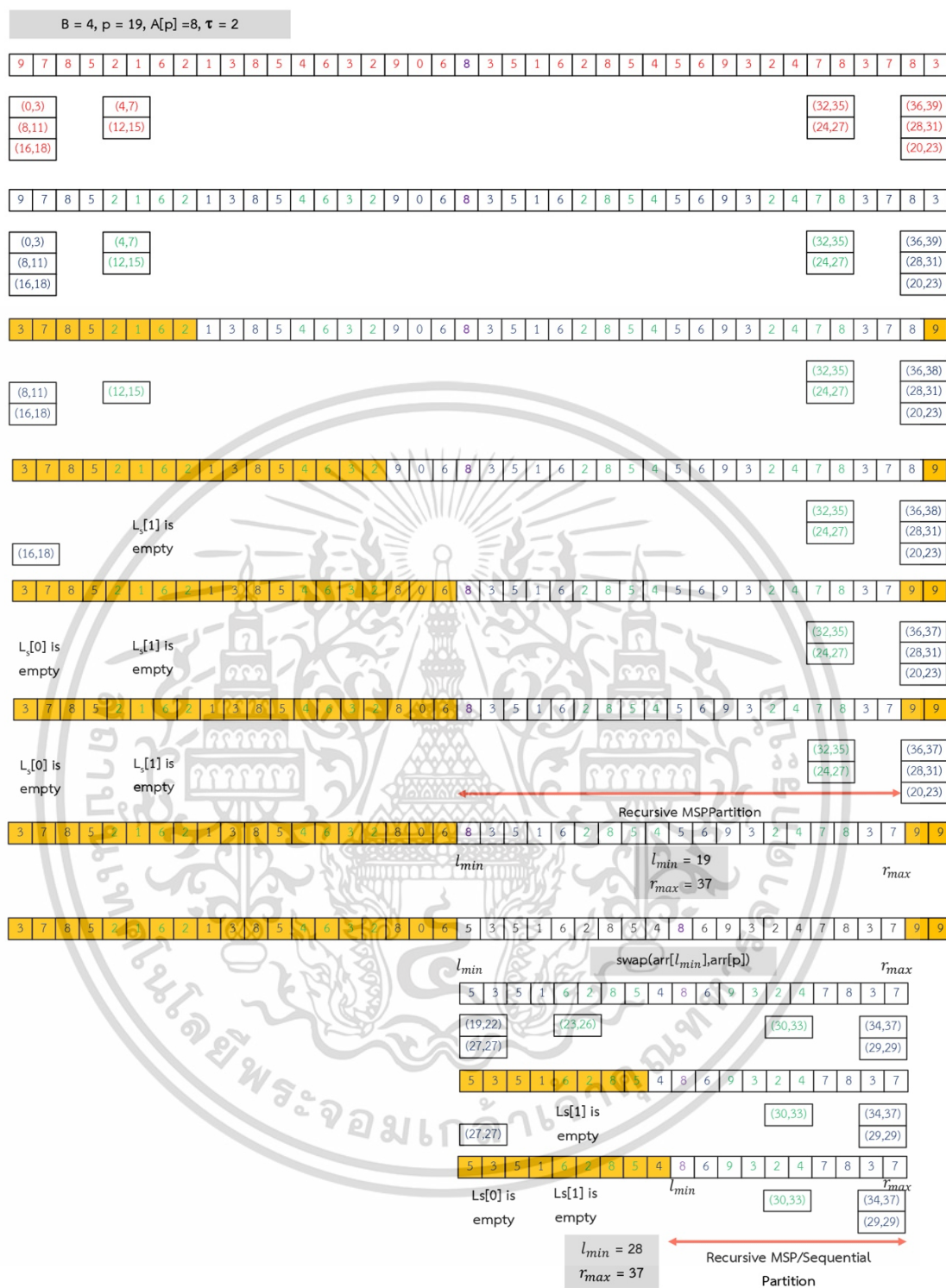
อัลกอริธึม	ตัวชี้วัด	20M	50M	100M	200M	200M $\beta=\text{ON}$
RAL-MSPSort	T_{Q3}	1.7501	4.6861	9.7268	20.5472	20.2142
	$\bar{T}$	1.7335	4.6198	9.6218	20.3233	20.0028
	T_{Q2}	1.7302	4.6215	9.5645	20.2944	19.9662
	T_{Q1}	1.7137	4.5675	9.4322	20.0442	19.7528
	T_{IOR}	0.0364	0.1186	0.2946	0.5030	0.4614
	σ_T	0.0328	0.1771	0.3427	0.6693	0.4716
	$V_{\text{usr}}(\%)$	87.3213	87.3098	86.9115	87.9475	87.9766
	$V_{\text{ram}}(\%)$	100.3140	100.2474	100.2008	100.1848	
BQSort	T_{Q3}	1.8262	4.8709	10.2529	21.6878	
	$\bar{T}$	1.7977	4.7990	10.0812	21.3060	
	T_{Q2}	1.7821	4.7602	10.0035	21.0510	
	T_{Q1}	1.7562	4.6768	9.8058	20.6554	
	T_{IOR}	0.0700	0.1941	0.4471	1.0324	
	σ_T	0.0593	0.1729	0.4180	1.0467	
	$V_{\text{usr}}(\%)$	92.0551	89.5123	86.6695	87.4219	
	$V_{\text{ram}}(\%)$	100.2882	100.1184	100.0588	100.0297	
MWSort	T_{Q3}	1.5818	4.0025	8.9359	18.2426	
	$\bar{T}$	1.5700	4.0276	8.7317	18.0441	
	T_{Q2}	1.5659	3.9595	8.8775	18.0956	
	T_{Q1}	1.5542	3.9384	8.7910	17.9187	
	T_{IOR}	0.0276	0.0641	0.1449	0.3239	
	σ_T	0.0244	0.1598	0.3418	0.4057	
	$V_{\text{usr}}(\%)$	86.3102	86.8115	82.9921	82.7472	
	$V_{\text{ram}}(\%)$	200.2374	200.1084	200.0543	200.0269	

ภาคผนวก ค

ค.1 ตัวอย่างการเกิดไพวอทเอียงใน MSPPartition

ภาพที่ ค.1 แสดงตัวอย่างการเกิดไพวอทเอียงใน MSPPartition ซึ่งการเกิดไพวอทเอียงนั้น เป็นสิ่งที่สามารถเกิดขึ้นได้ในอัลกอริธึมควิกซอร์ต หรืออัลกอริธึมที่ถูกพัฒนามาจากควิกซอร์ต โดยกำหนดให้ $B = 4$ ตำแหน่งไพวอท = 19 ค่าของไพวอท = 8 จำนวนเรด = 2 ตัวอย่างข้อมูลมีตั้งแต่ 0-9 เมื่ออัลกอริธึมเริ่มต้นจะมีการสร้างสแต็กด้านซ้ายและด้านขวาขึ้นมาอย่างละ 2 ชุดเท่ากับจำนวนเรด เพื่อเก็บขอบเขตของบล็อกที่ยังไม่ได้ทำ MSPPartition โดยข้อมูลของสแต็ก 0 จะใช้สีน้ำเงิน ข้อมูลของสแต็ก 1 ใช้สีเขียว (แถวที่ 2) จากนั้นทำ MSPPartition จนกระทั่งมีสแต็กใดสแต็กหนึ่งของแต่ละเรดว่าง (แถวที่ 3-6) จากภาพ $L_s[0]$ และ $L_s[1]$ ว่าง โดยสีส้มจะเป็นข้อมูลที่ทำการแบ่งข้อมูล โดยใช้ MSPPartition แล้ว ซึ่งจะเห็นว่าข้อมูลฝั่งซ้ายมีการแบ่งข้อมูลเรียบร้อยแล้ว แต่ด้านขวามีข้อมูลที่ถูกแบ่งเพียง 2 ข้อมูล (แถวที่ 6) จากนั้นอัลกอริธึมจะหาค่าตำแหน่งไพวอทใหม่ โดยหาค่าเฉลี่ยระหว่าง l_{min} และ r_{max} ซึ่งจากภาพจะได้ $(19+37)/2 = 28$ (แถวที่ 7) จากนั้นสลับข้อมูลตำแหน่งของ l_{min} และตำแหน่งใหม่ที่คำนวณได้ (แถวที่ 8) ทำเช่นนั้นวนซ้ำไปจนกระทั่งขนาดของอาร์เรย์ย่อยไม่มากไปกว่า u_{sort} (แถวที่ 9-11)

จากภาพที่ ค.1 จะเห็นว่า หากมีการเลือกไพวอทที่เอียงมาก จะทำให้การแบ่งข้อมูลสามารถทำได้เพียงด้านใดข้างหนึ่งเพียงด้านเดียว ซึ่งส่งผลให้มีการแบ่งข้อมูลสำเร็จเพียงครึ่งหนึ่งของอาร์เรย์ย่อยทั้งหมดในขั้นนั้น และจะต้องมีการเรียก MSPPartition ซ้ำหลายชั้นจนกระทั่งความยาวของอาร์เรย์ย่อยไม่มากกว่า u_{sort}



ภาพที่ ค.1 ตัวอย่างการเกิดไฟวอทเอียงใน MSPPartition

ภาคผนวก ง

ง.1 งานวิจัยที่ตีพิมพ์

มีงานวิจัยที่ตีพิมพ์ในวารสาร Journal of Mobile Multimedia 2 เรื่อง คือ
 A MultiStack Parallel (MSP) Partition Algorithm Applied to Sorting
 และ
 Optimizing MultiStack Parallel (MSP) Sorting Algorithm



เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
 ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

A MultiStack Parallel (MSP) Partition Algorithm Applied to Sorting

Apisit Rattanatanurak and Surin Kittitornkun*

*Dept. of Computer Engineering, Faculty of Engineering, King Mongkut's Institute
of Technology Ladkrabang, Bangkok, Thailand 10520*

E-mail: apisit.ra@ssru.ac.th; surin.ki@kmitl.ac.th

**Corresponding Author*

Received 19 May 2020; Accepted 24 June 2020;
Publication 08 September 2020

Abstract

The CPUs of smartphones are becoming multicore with huge RAM and storage to support a variety of multimedia applications in the near future. A MultiStack Parallel (MSP) sorting algorithm is proposed and named MSPSort to support manycore systems. It can be regarded as many threads of single-pivot interleaving block-based Hoare's algorithm. Each thread performs compare-swap operations between left and right (stacked and interleaved) data blocks. A number of multithreading features of OpenMP and our own optimization strategies have been utilized. To simulate those smartphones, MSPSort is fine tuned and tested on four Linux systems, e.g. Intel i7-2600, Xeon X5670, AMD R7-1700 and R9-2920. Their memory configurations can be classified as either uniform or non-uniform memory access. The statistical results are satisfied compared to parallel-mode sorting algorithms of Standard Template Library, namely Balanced QuickSort and MultiWay MergeSort. Moreover, MSPSort looks promising to be developed further to improve both run time and stability.

Keywords: Partition, Sort, Multithread, Parallel, OpenMP, Stack.

Journal of Mobile Multimedia, Vol. 16_3, 293–316.

doi: 10.13052/jmm1550-4646.1632

© 2020 River Publishers

1 Introduction

Manycore CPUs are prevalent in both servers and high-end desktop personal computers as uniform/non-uniform memory access (UMA/NUMA) systems. In the near future, smartphones' CPUs are becoming multicore towards manycore to support a variety of multimedia applications. Therefore, basic computing algorithms shall be adapted to exploit that. Sorting and data partitioning are mostly based on the well known single-pivot Hoare's algorithm. It is known as QuickSort divide and conquer (D&Q) behavior. The first level partition is the bottleneck of D&Q Hoare's algorithm. This paper intends to tackle this problem with multithreading techniques while minimizing the unnecessary memory accesses.

In this paper, we propose a single-pivot block-based data partition algorithm named MultiStack Parallel Partition (MSPPartition). As an application of MSPPartition, MSPSort is proposed to recursively divide the data array into shorter subarrays and to sort them in parallel. Unlike other block-based partitioning algorithms, MSPSort is based on stacks rather than queues and deques. Our contributions can be listed here. Firstly, the MSPSort is in-place and requires zero extra memory to buffer the partitioned data. Secondly, the parallel multistack compare-swap operation is similar to the sequential Hoare's algorithm thus demanding low memory bandwidth. Thirdly, a hybrid breadth-first depth-first task scheduling is proposed to support cache locality while maximizing parallelism.

This paper is organized as follows. Section 2 reviews related background and previous work of parallel D&Q sorting algorithms. The MSPPartition and MSPSort are elaborated in Section 3. Later on, experiment results are discussed in detail. The last section is Conclusions and Future Work.

2 Background and Related Work

This section consists of the following subsections, *Parallel Sorting Algorithms* and *STLSort: Sequential and Parallel Modes*.

2.1 Parallel D&Q Sorting Algorithms

In 1990, Heidelberg et al. [4] first presented simulation results of parallel QuickSort based on three parallel partitioning algorithms using Fetch-and-Add (F&A) operations and two scheduling algorithms. Speedup of $400\times$ can be obtained from sorting 2^{20} data with upto 500 processors, low-cost

F&A operations and other ideal assumptions. In 2003, Tsigas and Zhang [14] proposed a block-based parallel partitioning QuickSort algorithm. The block size is as small as the L1 cache which we consider it as fine-grained parallelism. Its speedup of $11\times$ can be achieved with 32 processors of SUN-T1 architecture. Süß and Leopold [12] presented several alternative algorithms of parallel QuickSort based on Pthread and OpenMP 2.0 in 2004. It can achieve $3.24\times$ on a 4-core AMD Opteron 848. In 2007, Singler et al. [11] developed Multi-Core Standard Template Library (MCSTL) based on C++ Standard Template Library. This parallel sorting algorithm is similar to Tsigas and Zhang's [14] with a double-ended queue (deque). Its Speedup of $21\times$ can be achieved on an 8-core 32-thread SUN-T1.

In 2008, Traoré et al. [13] described work-optimal parallelizations of STL sort based on work-stealing technique. However, their Introspective sort based on parallel block-based partition [8], [15] is deque-free. Speedup of $8.1\times$ with 16 processors can be obtained. One year later in 2009, Ayguadé et al. [2] proposed MultiSort based on MergeSort which splits the input data equally, sorts them using QuickSort in parallel and then merges them using OpenMP 3.0 Task construct. A maximum Speedup of $13.6\times$ on 32 cores can be achieved with Intel's C++ Compiler version 9.1 and Cilk compiler version 5.4.3 using last in first out software thread queue. Meanwhile, Man et al. [6, 7] developed *psort()*, a hybrid QuickSort and MergeSort algorithm. Their work can achieve $11\times$ Speedup on a 24-core Intel Xeon 7460 system.

In 2013, Mahafzah [5] split the input array with multi-pivot/threads into partitions using extra space and then sort them in parallel with 8 software threads. Speedup of $3.8\times$ is achieved on a dual-core HyperThread processor. Later on, Ranokpanuwat and Kittitornkun [9] proposed Parallel Partition and Merge QuickSort (*PPMQSort*). They can achieve Speedup of $12.29\times$ relative to *qsort()* on an 8-core HyperThread Xeon E5520 in 2016. More recently in 2017, Axtmann et al. [1] presented an IPS^4 sorting algorithm. It is a recursive multithread in-place bucket sort. Each thread is responsible for classifying a number of data blocks into local k buckets based on multipivot values. The local buckets are merged to replace the input array. Once the merged subarrays are shorter and then sorted independently. Speedup can be as high as $29\times$ over its sequential version on a 32-core Intel Xeon E5-2683 v4. In 2018, Rattanatanurak [10] proposed parallel sorting named Dual Parallel Partition sorting (*DPPSort*). Speedups of $5.95\times$ and $4.70\times$ can be achieved relative to *qsort()*, and *STLSort*, respectively on 4-core HyperThread Intel i7-3770. In summary, Table 1 compares some parallel sorting

Table 1 Comparison of Sorting Algorithms in terms of Partition Granularity, Merge Algorithm, Time Complexity and Library in chronological order (*BQSort*: Balanced QuickSort, *MWSort*: MultiW Merge Sort, *DFWSort*: Deque-Free Work-Optimal Parallel *STLSort*, *PMQSort*: Parallel Multithreaded QuickSort, *PPMQSort*: Parallel Partition and Merge QuickSort, *IPS⁴o*: In-Place Parallel Super Scalar Sample Sort, *DPPSort*: Dual Parallel Partition Sort (B-neck: Bottleneck, Seq: Sequential, NA: Not Available, N : Array Size, c : CPU cores)

Algorithm	[14](2003)	[11](2007)
Name	<i>PQuickSort</i>	<i>BQSort</i>
Granularity	Fine: L1 Cache	Fine: L1 Cache
B-neck	Seq Swap to Middle	Swap to Middle
Recursive	Yes	Yes
Time	$O(\frac{N}{c} \log \frac{N}{c} + \frac{N}{c})$	$O(\frac{N}{c} \log N + c \log c)$
Library	Pthread	OpenMP
Algorithm	[11](2007)	[13](2008)
Name	<i>MWSort</i>	<i>DFWSort</i>
Granularity	NA	Fine: L1 Cache
B-neck	pW merging	Swap to Middle
Recursive	Yes	Yes
Time	$O(\frac{N}{c} \log N + (c \log c)(\log \frac{N}{c}))$	$O(\frac{N}{c} + \log^3 N)$
Library	OpenMP	OpenMP
Algorithm	[6](2009)	[9](2016)
Name	<i>psort</i>	<i>PPMQSort</i>
Granularity	Coarse: N/c	Coarse: $N/2$
B-neck	Seq Merge then <i>qsort</i>	Seq Swap
Recursive	No	Yes
Time	$O(\frac{N}{c} \log \frac{N}{c} + N)$	$O(\frac{N}{c} \log \frac{N}{2c} + N)$
Library	OpenMP	OpenMP
Algorithm	[1](2017)	[10](2018)
Name	<i>IPS⁴o</i>	<i>DPPSort</i>
Granularity	Fine: block-based	Coarse: $N/2$
B-neck	In-place buckets	Partition then Swap
Recursive	Yes	Yes
Time	NA	$O(\frac{N}{c} \log \frac{N}{2c} + N)$
Library	OpenMP	OpenMP

algorithms in chronological order such as partition granularity, bottleneck, recursion, Big-O time complexity and parallel library.

2.2 STLSort: Sequential and Parallel Modes

The Standard Template Library (*STL*)Sort is a sequential sorting function for any data type. It is available in almost C++ compilers and prototyped as follow.

```
std::sort(RandomAccessIterator first, RandomAccessIterator
          last);
```

Parameters *first* and *last* are pointers to the first and the last positions, respectively. On the other hand, GNU libstdc++ parallel mode [11] provides two parallel sorting functions based on OpenMP. Namely, Balanced QuickSort and Multiway Merge Sort, are subject to evaluation in our experiments. Its function is declared in `<parallel/algorithm>` directive as follow.

```
__gnu_parallel::sort(RandomAccessIterator first,
                     RandomAccessIterator last);
```

2.2.1 Balanced QuickSort (BQSort)

BQSort is block based similar to Tsigas and Zhang's [14] partition method. It compares/swaps data between pairs of left and right blocks in parallel until either side is finished. The unfinished (leftover) data blocks are pushed to a double ended queue (deque) to process later. As a result, a pair of blocks can be stolen to any free processor core. The unfinished blocks are swapped to the middle of the input array so that the array can be eventually partitioned. Sequential *STLSort* is executed locally after it is partitioned successfully. It is claimed to be an in-place algorithm which can be load-balanced using Work Stealing method. Run time of this algorithm is varied depending on data distribution.

2.2.2 MultiW Merge Sort (MWSort)

MWSort divides data into several subarrays equally and *STLSort* them in parallel. Each subarray is sorted independently with small overheads. *MWSort* relies on parallel multiway merging algorithm to obtain the final data array. Subsequently, the sorted temporary array is copied to the input array. As a result, this *MWSort* requires at least twice the space of input data size. Its run time is stable compared with quicksort algorithm.

3 MultiStack Parallel Sort (MSPSort)

This section begins with the overview of our algorithm consisting of the *Recursive MultiStack Parallel Partition* and *Sorting* Phases. Consecutively, a number of BF-DF Scheduling algorithms are proposed and compared.

In the **MSPSort()** function, Median of Five function *MO5()* (Alg. 1, line 5) selects a pivot index *p* and moves it to the middle of array *A*. The

Algorithm 1: MSPSort Algorithm

```

1  Function Main()
2  |   MSPSort( $A, 0, N - 1, \tau_{max}$ )  // MSPSort array  $A$  with  $\tau_{max}$  threads
3  EndFunction
4  Function MSPSort( $A, i_L, j_R, \tau$ )
5  |    $p \leftarrow MO5(A, i_L, j_R)$  //  $p$ =Median of Five
6  |   if  $j_R - i_L > u_{stl}$  then
7  |   |    $p \leftarrow MSPPartition(A, i_L, j_R, p, \tau)$  // with  $\tau$  threads
8  |   |   if  $\tau > \tau_{max}/r$  then
9  |   |   |    $\tau \leftarrow \tau/2$  // Reduce  $\tau$  threads by 2
10 |   |   end
11 |   |   if  $j_R - i_L > u_{df}$  then
12 |   |   |   BFMSPSort( $A, i_L, j_R, p, \tau$ ) // Breadth First with  $\tau$  threads
13 |   |   end
14 |   |   else
15 |   |   |   DFMSPSort( $A, i_L, j_R, p, \tau$ ) // Depth First with  $\tau$  threads
16 |   |   end
17 |   end
18 |   else
19 |   |   OpenMP Task
20 |   |   STLSort( $A + i_L, A + j_R$ ) // Call STLSort as a task
21 |   |   OpenMP nowait
22 |   end
23 EndFunction
24 Function BFMSPSort( $A, i_L, j_R, p, \tau$ )
25 |   OpenMP Task
26 |   MSPSort( $A, i_L, p - 1, \tau$ ) // left subarray  $\tau$  threads
27 |   OpenMP Task
28 |   MSPSort( $A, p + 1, j_R, \tau$ ) // right subarray  $\tau$  threads
29 EndFunction
30 Function DFMSPSort( $A, i_L, j_R, p, \tau$ )
31 |    $P_s.push(i_L, j_R)$  // Push the partition's boundary
32 |   while  $P_s$  not empty do
33 |   |    $i_L, j_R \leftarrow P_s.pop()$  // Pop the partition's boundary
34 |   |   if  $j_R - i_L > u_{stl}$  then
35 |   |   |    $p \leftarrow MO5(A, i_L, j_R)$  //  $p$ =Median of Five
36 |   |   |    $p \leftarrow MSPPartition(A, i_L, j_R, p, \tau)$  // with  $\tau$  threads
37 |   |   |    $P_s.push(i_L, p - 1)$  // Push the left boundary
38 |   |   |    $P_s.push(p + 1, j_R)$  // Push the right boundary
39 |   |   end
40 |   |   else
41 |   |   |   OpenMP Task
42 |   |   |   STLSort( $A + i_L, A + j_R$ ) // Call STLSort as a thread
43 |   |   |   OpenMP nowait
44 |   |   end
45 |   end
46 EndFunction

```

Algorithm 2: Parallel Stacked Blocks Partition

```

1  Function MSPPartition( $A, i_L, j_R, p, \tau$ )
2       $halfB \leftarrow (j_R - i_L) / (2b)$  // Number of blocks on each side
3      for  $i \leftarrow 0$  to  $halfB - 1$  do
4           $L_s[i \bmod \tau].push(i_L + i, i_L + i + b)$  // Push left blocks
5           $R_s[i \bmod \tau].push(j_R - i - b, j_R - i)$  // Push right blocks
6           $i \leftarrow i + 1$ 
7      end
8      begin OpenMP parallel for  $private(i, j, l_b, r_b)$ 
9      for  $t \leftarrow 0$  to  $\tau - 1$  do
10         while  $L_s[t]$  not empty &&  $R_s[t]$  not empty do
11              $(i, l_b) \leftarrow L_s[t].pop()$  // Pop left top block boundary
12              $(r_b, j) \leftarrow R_s[t].pop()$  // Pop right top block boundary
13             do
14                 while  $A[i] \leq A[p]$  &&  $i \leq l_b$  do
15                      $i \leftarrow i + 1$  // Increase i index
16                 end
17                 while  $A[j] > A[p]$  &&  $j \geq r_b$  do
18                      $j \leftarrow j - 1$  // Decrease j index
19                 end
20                 if  $i \leq l_b$  &&  $j \geq r_b$  then
21                      $SWAP(A[i], A[j])$  // Swap  $A[i]$  and  $A[j]$ 
22                      $i \leftarrow i + 1$  // Increase i index
23                      $j \leftarrow j - 1$  // Decrease j index
24                 end
25                 while  $i \leq l_b$  &&  $j \geq r_b$ 
26                 if  $i > l_b$  then
27                      $R_s[t].push(r_b, j)$  // Push the right block boundary
28                     back
29                 end
30                 else if  $j < r_b$  then
31                      $L_s[t].push(i, l_b)$  // Push the left block boundary back
32                 end
33             end
34              $l_{min} \leftarrow \min(L_s[t], \forall t)$  // Find the left most index
35              $r_{max} \leftarrow \max(R_s[t], \forall t)$  // Find the right most index
36              $\mu \leftarrow (r_{max} - l_{min}) / u_{stl}$  // Threads to deal with the middle one
37             if  $r_{max} - l_{min} > u_{stl}$  then
38                 return MSPPartition( $A, l_{min}, r_{max}, p, \mu$ ) // With  $\mu$  threads
39             end
40             else
41                 return LomutoPartition( $A, l_{min}, r_{max}, p$ ) // Lomuto's Partition
42             end
43 EndFunction

```

Recursive MSPartition partitions the input array A according to the pivot and finally returns the position of pivot p (Alg. 1, line 7). MSPSort continues according to our proposed scheduling (Alg. 1, lines 12 and 15). The resulting shorter than u_{stl} subarray is sorted as an independent thread (Task) (Alg. 1, line 20) using STLSort where $u_{stl} = U_{stl} \times \kappa_{l3}/sizeof(Type)$, U_{stl} is Sorting Cutoff parameter, κ_{l3} represents the Level 3 cache size and $Type$ corresponds to the data type to be sorted. Note that, the number of software threads τ is reduced to $\tau/2$ (Alg. 1, line 9) and remained at $\tau = \tau_{max}/r$ in order to balance the workload and achieve parallelism where τ_{max} is the maximum number of threads and r is called Reduction factor.

3.1 Recursive MultiStack Parallel Partition Phase

The *Recursive MultiStack Parallel Partition Phase* consists of 2 steps: *Parallel Stacked Blocks Partition Step* and *Middle Blocks Partition Step*.

The *Parallel Stacked Blocks Partition Step* begins with dividing $A = A[0], A[1], \dots, A[N-1]$, an unsorted array into left and right halves. Each half is divided into blocks of $b = B \times \kappa_{l3}/sizeof(Type)$ elements from both ends (Alg. 2, line 4) where B is a block size parameter. Both left and right block boundaries on the both halves are assigned in round robin to τ threads and pushed from the middle towards both ends (Alg.2, lines 4 and 5). Therefore, each thread is assigned with about the same number of blocks to manipulate and balance the workload while achieving parallelism simultaneously.

When the stacks are ready, OpenMP **parallel for** is applied to fork τ threads (Alg. 2, line 8) with private (local to each thread) variables i, j, l_b, r_b . Subsequently, these block boundaries are popped off so that data within the left and right blocks can be compared with $A[p]$ and swapped from both ends to the middle until either local left or right stack is empty (Alg. 2, lines 10). Each thread has its own private variables i and j that are left and right indices of the current left and right blocks, respectively. In addition, variables, l_b and r_b are the current boundaries of left and right blocks, respectively. Eventually, the boundaries of the unfinished block are pushed back to their corresponding stacks (Alg.2, lines 27 and 30). This step stops when all τ threads finish.

After that, two indices, $l_{min} = \min(L_s[t], \forall t)$ and $r_{max} = \max(R_s[t], \forall t)$ of all τ threads, must be determined to compute $r_{max} - l_{min}$ whether the leftover part is longer than u_{stl} (Alg. 2, line 37). In *Middle Blocks Partition Step*, the length of the leftover can indicate the number of μ threads to call **MSPPartition()** (Alg. 2, line 38) just in case. Otherwise, the Lomuto's

Partition [3] eventually returns the pivot index p (Alg. 2, line 41). That is because Lomuto's algorithm requires fewer memory accesses than Hoare's.

3.2 Sorting Phase

In the earlier phase, the data subarray is partitioned into smaller subarrays recursively. Any shorter subarray up to u_{stl} elements can be sorted using *STLSort* as a independent task (Alg. 1, lines 20 and 42) without any synchronization (OpenMP *nowait*).

3.3 BF-DF Scheduling Algorithms

The *Recursive MSPPartition Phase* initially employs default scheduling of OpenMP and thus called BF (Breadth First) method to achieve high parallelism. The problem of BF scheduling is due to its random order of executions depending on the partition sizes and branch/memory stalls. This may cause unnecessary page faults and cache misses. To avoid this problem, we have proposed and implemented DF (Depth First) sorting algorithm in **DFMSPSort()** function. Once enough number of tasks are queued up in the thread pool by BF algorithm, the partitioning process is continued in DF order.

Initially, if the subarray $(j_R - i_L)$ is still greater than u_{df} elements (Alg. 1, line 11), **BFMSPSort()** is called recursively (Alg. 1, line 12) as two OpenMP tasks. In other words, **BFMSPSort()** is executed recursively and continued until the resulting subarray is smaller than u_{df} elements where $u_{df} = U_{df} \times \kappa_{13} / \text{sizeof}(\text{Type})$ and U_{df} is Scheduling Cutoff. Otherwise, the alternative **DFMSPSort()** function is invoked instead (Alg. 1, line 15).

On line 32 of Alg. 1, a local stack P_s is instantiated to keep the subarray boundaries and enforce the execution order so that last-level cache misses can be minimized. Programmers can easily implement the DF scheduling by themselves without worrying about OpenMP supports. It makes use of a local stack P_s to keep the subarray boundaries. This stack can order the execution with one of these scheduling algorithms, RAL, LAL, SPF and LPF, to improve cache locality.

3.3.1 RAL vs. LAL

First of all, the first partition is pushed onto the stack P_s (Alg. 1, line 31). The popped off indices i_L, j_R are passed to *Recursive MSPPartition Phase* (Alg. 1, line 36). Once the left and right subarrays are obtained, the boundaries of the left one are pushed prior to the right one resulting to depth first

traversal to the right hand side (Right Always: RAL). The *Recursive MSPP Phase* continues until the subarray is shorter than u_{stl} . Note that *STLSort* is executed independently with **OpenMP nowait** compiler directive (Alg. 1, line 43). The traversal continues until P_s is empty (Alg. 1, line 32). The LAL (Left Always) algorithm is the opposite of RAL.

3.3.2 SPF vs. LPF

Both RAL and LAL algorithms make the decisions based on the direction only regardless of the subarray size. It can be more beneficial to our MSPPartition if cache replacement policy is taken into consideration. The shorter partition first (SPF) and longer partition first (LPF) decide longer or shorter subarray to push onto the stack first, respectively. As such, the SPF decision may exploit more recently accessed data inside the caches. On the other hand, the LPF one may prefer longer workload to sustain parallelism.

4 Experiments, Results and Discussions

This section presents how to set up the experiments on four different Linux systems. Experiment parameters are listed and rationalized. Consecutively, the obtained results are elaborated and discussed.

4.1 Experiment Setup

The proposed MSPSort algorithm is evaluated on four different systems as listed in Table 2. They all run the same Ubuntu 18.04 LTS and G++ version 7.4.0. Both Intel and AMD processors are provided equally and subject to our resource constraints. The number of cores c is reported by Linux System Monitor. Moreover, these systems widely differ in terms of memory size, technology and configuration. Nonetheless, their caches are quite similar. Most of their L3 caches are multiples of 8MB that we use κ_{l3} to denote. Note that NUMA stands for non-uniform memory access time. R7-1700 consists of two memory controllers, one on each die and interconnected with the Infinity Fabric. That results in non-uniform memory latency [www.tomshardware.com].

The experiments are parameterized as shown in Table 3. The data types to be evaluated include Unsigned 32-bit integer (Uint32), Unsigned 64-bit integer (Uint64) and 64-bit double precision floating point numbers (Double). They are randomized with uniform distribution. All algorithms are optimized

Table 2 Specifications of multicore CPUs in experiments, KB: Kilobytes, MB: Megabytes

Series Number	Core i7 i7-2600	Xeon X5670	Ryzen R7-1700	ThreadRipper R9-2920
Clock (GHz)	3.40	2.93	3.00	3.50
c (cores)	8	24	16	24
Sockets	1	2	1	1
RAM	32GB	24GB	32GB	64GB
Configuration	4×8GB	12×2GB	4×8GB	8×8GB
Technology	DDR3	DDR3	DDR4	DDR4
NUMA	No	Yes	Almost	Yes
Memory	2 ch	4 ch	2 ch	4 ch
L1 I-Cache	4×32KB 8W	2×6×32KB 4W	8×64KB 4W	12×64KB 4W
L1 D-Cache	4×32KB 8W	2×6×32KB 8W	8×32KB 8W	12×32KB 8W
L2 Cache	4×256KB 8W	2×6×256KB 8W	8×512KB 8W	12×512KB 8W
L3 Cache	8MB 16W	2×12MB 16W	2×8MB 16W	4×8MB 16W

Table 3 Experiment parameters of *MSPSort*, BF: Bread-First, DF: Depth-First, $M=10^6$

Parameters	Values
Algorithms	<i>MSPSort</i> , <i>BQSort</i> , <i>MWSort</i>
Data Types	Uint32, Uint64, Double
Random Dist	Uniform
GCC Optimization	O2
Data size N	200M, 500M, 1000M, 2000M
Scheduling	RAL, LAL, LPF, SPF
L3 Cache size κ_{l3}	8MB
Block size $B(\times\kappa_{l3})$	10^{-4} , 10^{-3} , 10^{-2} , 10^{-1} , 1
Cutoff $U_{stl}(\times\kappa_{l3})$	0.5, 1, 2, 4, 8
Cutoff $U_{df}(\times\kappa_{l3})$	0.5, 1, 2, 4, 8, 16
Multiplier m	1, 2, 4
Reduction r	c , $c/2$, $c/3$, $c/4$

with -O2 compiler flag. The data size N ranges from 200M to 2000M elements due to system RAM limit. Our proposed BF-DF scheduling can be chosen among these algorithms, LPF, SPF, RAL and LAL.

As mentioned earlier, the block size B , Sorting Cutoff U_{stl} and Scheduling Cutoff U_{df} are functions of L3 Cache size $\kappa_{l3}=8\text{MB}$. The block size $B=10^{-4}$, 0.001, 0.01, 0.1, 1. Sorting Cutoff $U_{stl} = 0.5, 1, 2, 4$. Scheduling Cutoff $U_{df} = 1, 2, 4, 8, 16$. The Multiplier m is set to be power of two, $m = 1, 2, 4$ as such the MSPSort can fork as many $\tau_{max} = c \times m$ threads. The Reduction r can be formulated as a function of c cores reported by the OS, $r = c, c/2, c/3, c/4$.

4.2 Key Performance Indicators (KPIs)

In this paper, some experiment results shall be normalized and compared based on these KPIs. They all represent time domain aspects of each sorting algorithm.

4.2.1 Average Run Time ($\bar{T}$) and Run Time per 100M ($\bar{T}_{100M}$)

The Average Run Time ($\bar{T}$) is averaged over a number of trials as specified in each experiment. The proposed Run Time per 100M ($\bar{T}_{100M}$) is easy to visualize and compare at any data size for certain experiments. In addition, this normalized run time can enable comparison between systems.

4.2.2 Standard Deviation of T (σ_T) and T_{100M} (σ_{100M})

Run Time Standard of Deviation (σ_T) represents the stability of each algorithm due to the randomness of generated data set. In addition, the normalized standard deviation (σ_{100M}) can justify some parameters specially Block size B and U_{stl} .

4.2.3 Run Time Statistics

In addition to arithmetic mean and standard deviation of of Run Time T , the first, second and third quartiles are T_{Q1} , T_{Q2} and T_{Q3} , respectively. In addition, InterQuartile Range can be determined as $T_{IQR}=T_{Q3}-T_{Q1}$ for stability analyses. These statistics can specify how the Run Time T distributes over 1,000 trials.

4.3 Single-Round MSPPartition

This single round MSPPartition is a prerequisite experiment as a guidance to the main ones. In order to fine tune block size B , a simple partition is tested at various block sizes as listed in Table 3. This experiment is intended to investigate Block size B effects of MSPPartition (Alg. 2, Line 1). Within this experiment, data within left and right blocks are always swapped to get rid of branch prediction (comparison) effects, Given a data array size N , Function MSPPartition is executed for just one round without further recursive calls. The Block size B in this experiment spans a wide range, $\{10^{-4}, 0.001, 0.01, 0.1, \text{ and } 1\} \times \kappa_{l3}$ cache size. The maximum number of threads $\tau_{max} = c \times 1$. Note that OpenMP nested parallelism flag is turned off, `omp_set_nested(0)`.

The resulting $\bar{T}_{100M}$ (bar) and $\pm\sigma_{100M}$ (error bar) in seconds are plotted in Figure 1 at different data sizes after 100 trials. All systems show the same behavior of $\bar{T}_{100M}$ vs B . It can also be observed that the larger the data size

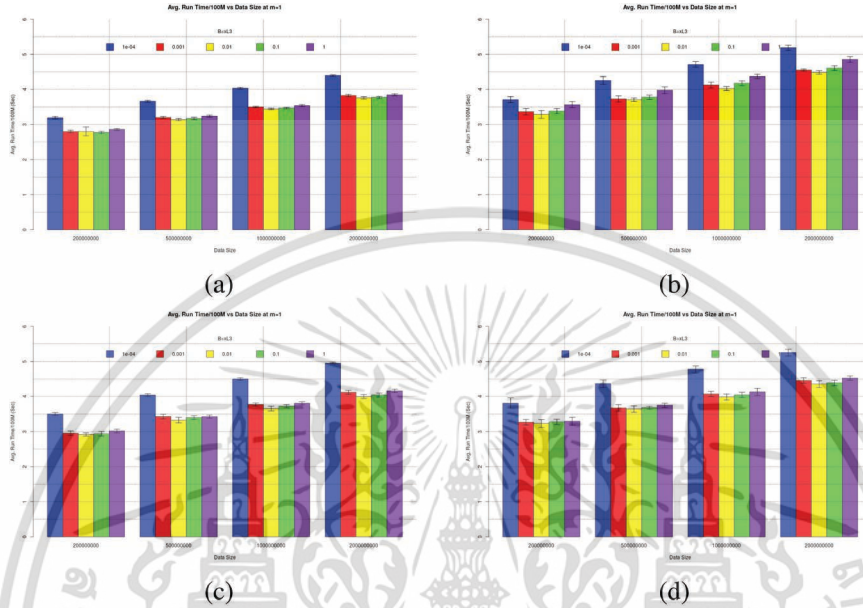


Figure 1 $\bar{T}_{100M}$ (Bargraph) and $\pm\sigma_{100M}$ (Error bar) of Single-Round MSPPartition at $B=\{10^{-4}, 0.001, 0.01, 0.1, 1\} \times \kappa_{l3}$, $m=1$ (a) i7-2600, (b) X5670, (c) R7-1700, (d) R9-2920 for Uint32 data and 100 trials

N , the higher the $\bar{T}_{100M}$. This can be due to poor cache locality accessing data from both ends. The smallest $B=10^{-4} \times \kappa_{l3} \approx 800$ Bytes yields the worst performance. The best $\bar{T}_{100M}$ can be found as B ranges between 0.001 to 0.1 $\times \kappa_{l3}$ that is between the size of L1 and L2 caches. As a result, $B = 0.01 \times \kappa_{l3}$ is chosen as a representative.

Note that all graphs are plotted on the same scale of Y axis. With $m=1$, each system gets different number of threads c to execute. That means i7-2600 can achieve lower $\bar{T}_{100M}$ on the same N than X5670 despite much lower core count. Similarly, R7-1700 yields faster $\bar{T}_{100M}$ than R9-2920 despite lower clock frequency and lower core count. This phenomenon could be due to non-uniform (longer) memory access of large data arrays on X5670 and R9-2920 as listed in Table 2.

4.4 Parallel Sorting of Independent Data Blocks

To investigate how Sorting Cutoff U_{stl} affects the Run Time, a data array of N elements is divided with equal chunks of u_{stl} elements and assigned to a

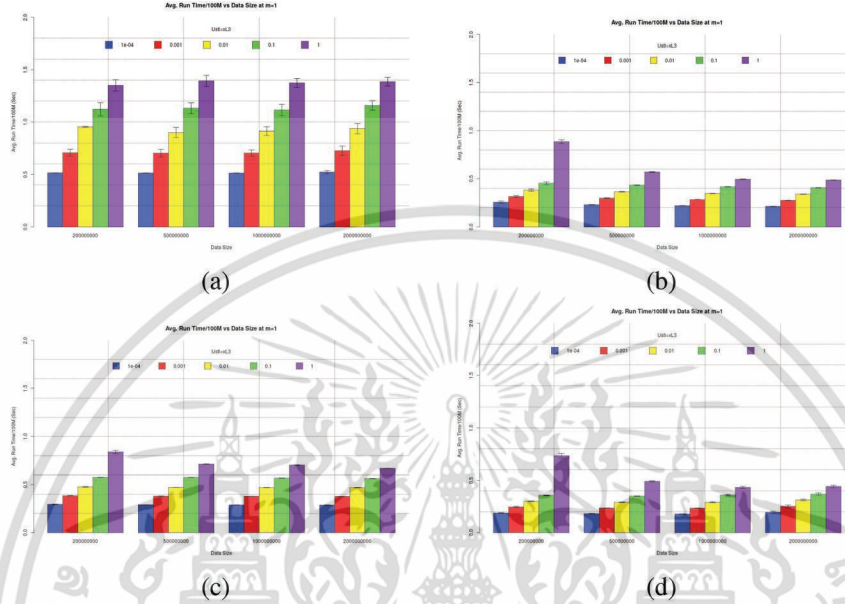


Figure 2 $\bar{T}_{100M}$ (Bargraph) and $\pm\sigma_{100M}$ (Error bar) of Independent Parallel Sort at $U_{stl}=\{10^{-4}, 0.001, 0.01, 0.1, 1\} \times \kappa_{l3}$, $m=1$ (a) i7-2600, (b) X5670, (c) R7-1700, (d) R9-2920 for Uint32 data and 100 trials

thread to sort in parallel. Divided subarrays are independently STLsorted with $c \times 1$ threads as $m=1$. Note that OpenMP nested parallelism flag is turned off just like the previous experiment. This experiment can be beneficial to any D&Q sorting algorithm in general because the partitioning overhead is neglected. The random data array of a given size N is divided equally to $U_{stl}=\{10^{-4}, 0.001, 0.01, 0.1, 1\} \times \kappa_{l3}$.

The experiment is repeated for 100 trials to obtain $\bar{T}_{100M}$ (bar) and σ_{100M} (error bar) as plotted in Figure 2. In general, the same behavior can be observed for all systems. It can be noticed that given the same data size N the smaller cutoff U_{stl} the lower $\bar{T}_{100M}$. This can be concluded that smaller U_{stl} is better provided that there is no dependency between these data chunks.

4.5 MSPSort with BF Scheduling

The current and later experiments are different from the preliminary ones where OpenMP Nested Parallelism is switched ON and MSPPartition is recursively invoked. MSPSort with BF scheduling corresponds to line 12 of Alg. 1 and line 11 is always true. Due to an extremely large number of

Table 4 Top-three (m,r) pairs with BF Scheduling for all N 's, $B=0.01$, $U_{stl}=0.5, 1, 2, 4$ after 20 Trials

System	i7-2600	X5670	R7-1700	R9-2920
Uint32	(2,8)	(1,6)	(2,16)	(2,12)
	(1,8)	(2,12)	(2,8)	(1,8)
	(2,4)	(1,8)	(1,16)	(1,6)
Uint64	(2,8)	(1,6)	(2,16)	(1,8)
	(1,4)	(2,12)	(1,8)	(1,6)
	(2,4)	(1,8)	(2,8)	(2,12)
Double	(2,8)	(1,6)	(2,16)	(1,8)
	(1,4)	(2,12)	(1,8)	(1,6)
	(2,4)	(1,8)	(2,8)	(2,12)

parameter combinations, this experiment is intended to obtain and pick (m,r) pair with the most consistent performance for each system. Run Time T 's are collected according with BF Scheduling for all N 's, $B=0.01$, $U_{stl}=0.5, 1, 2, 4$ after 20 Trials. The (m,r) pairs with most appearances in Top-10 minimum $\bar{T}$ of all data size N are listed in Table 4. The most consistent (m,r) pairs (top row of each data type) in Table 4 are selected for each system/data type as representatives for the next experiment.

4.6 MSPSort with BF-DF Scheduling

This experiment is intended to obtain the most consistent performance of (U_{stl}, U_{df}) pair and BF-DF scheduling algorithm given each data size N as listed in Table 5 for each system after 100 trials. For all data types, it can be observed that the (m,r) pairs are almost the same on many systems except R9-2920. It is not guaranteed that these parameters can yield consistent performance. Therefore, extensive run time statistics should be collected and compared against BQSort and MWSort.

Table 6 to Table 9 tabulates the run time statistics of all sorting algorithms after 1000 trials. According to the chosen parameters in Table tb:para:chosen, the time-domain KPIs of MSPSort can be investigated analyzed thoroughly. Although lower $\bar{T}$ and σ_T are better in terms of run time and stability, other statistics play important roles as well. We shall discuss the experiment results with respect to the following aspects.

4.6.1 Sorting vs Scheduling Cutoffs

There are two different approaches of BF-DF scheduling, direction versus size oriented. Both RAL and LAL are direction oriented. On the contrary,

Table 5 Chosen parameters $U_{stl}:U_{df}:m:r$, $B=0.01$ after 100 trials

System	i7-2600	X5670	R7-1700	R9-2920
UInt32				
BDFD	LPF	SPF	SPF	SPF
$N=200M$	0.5:2:2:8	0.5:1:1:6	0.5:1:2:16	0.5:1:2:12
$N=500M$	0.5:2:2:8	1:4:1:6	1:2:2:16	1:2:2:12
$N=1000M$	0.5:2:2:8	1:8:1:6	2:4:2:16	1:4:2:12
$N=2000M$	4:8:2:8	2:16:1:6	2:4:2:16	2:4:2:12
UInt64				
Double				
BDFD	RAL	RAL	LAL	LAL
$N=200M$	1:8:2:8	2:4:1:6	1:2:2:16	0.5:2:1:8
$N=500M$	1:8:2:8	2:4:1:6	1:2:2:16	1:4:1:8
$N=1000M$	2:8:2:8	4:8:1:6	2:2:2:16	2:8:1:8
$N=2000M$	2:8:2:8	4:8:1:6	2:2:2:16	2:8:1:8

LPF and SPF are size oriented. SPF and LPF are good for small data type such as UInt32. It can be also noticed that they mostly are characterized by smaller (U_{stl}, U_{df}) pairs. On the other hand, LAL and RAL are beneficial to MSPSort on larger data types such as both UInt64 and Double. The (U_{stl}, U_{df}) pairs are generally larger than those of UInt32.

As shown in Figures 1 and 2, all systems behave in the same fashion. It can be noticed in Figure 1 that $\bar{T}_{100M}$ significantly increases as N doubles up for all systems. Unlike partitioning $\bar{T}_{100M}$, sorting $\bar{T}_{100M}$ is almost constant for all data sizes N given the same U_{stl} . That means sorting can be traded off with partitioning at larger N as the subarrays become shorter.

In order to minimize the Run Time T , BD-DF Cutoff U_{df} grows according to N to reduce the recursion levels. We have showed in Figure 1 that partitioning $\bar{T}_{100M}$ is significantly higher as N doubles. Sorting cutoff U_{stl} is quite similar to U_{df} . It can be observed that U_{stl} is proportional to U_{df} as well. That is because sorting $\bar{T}_{100M}$ grows slowly as U_{stl} is ten fold longer in Figure 2. Therefore, sorting a longer subarray can take the same amount of time as partitioning it and sorting two resulting shorter subarrays.

4.6.2 Memory Architecture

Compared to BQSort only, MSPSort can achieve better run time statistics on all data types on every system except X5670. This can be due to the fact that BQSort can steal the workloads to distribute to available CPU cores. Thus, BQSort is more tolerant to multi-socket NUMA effects than MSPSort.

Table 6 Statistics of Run Time T of MSPSort vs BQSort vs MWSort for all data types at various sizes N on i7-2600 system after 1000 trials

Alg.	KPI (Sec.)	200M	500M	1000M	2000M
Uint32					
MSPSort	T_{Q1}	3.042	8.113	17.832	37.880
	T_{Q2}	3.073	8.179	17.963	38.340
	$\bar{T}$	3.182	8.342	17.928	38.332
	T_{Q3}	3.318	8.283	18.039	38.797
	σ_T	0.196	0.436	0.171	0.536
BQSort	T_{Q1}	3.212	8.578	18.285	38.670
	T_{Q2}	3.247	8.665	18.447	39.105
	$\bar{T}$	3.348	8.856	18.484	39.503
	T_{Q3}	3.491	8.804	18.663	40.130
	σ_T	0.198	0.503	0.270	1.111
MWSort	T_{Q1}	3.649	9.550	20.132	40.920
	T_{Q2}	3.700	9.675	20.382	41.588
	$\bar{T}$	3.812	9.710	20.385	41.764
	T_{Q3}	4.016	9.812	20.633	42.470
	σ_T	0.231	0.266	0.383	1.105
Uint64					
MSPSort	T_{Q1}	3.648	9.855	21.540	44.592
	T_{Q2}	3.772	9.909	21.725	44.813
	$\bar{T}$	3.813	9.956	21.712	44.887
	T_{Q3}	4.065	9.973	21.893	45.104
	σ_T	0.205	0.234	0.243	0.454
BQSort	T_{Q1}	3.702	10.023	21.780	45.898
	T_{Q2}	3.767	10.104	21.976	46.332
	$\bar{T}$	3.877	10.254	21.977	46.511
	T_{Q3}	4.151	10.233	22.144	47.073
	σ_T	0.227	0.442	0.270	0.776
MWSort	T_{Q1}	4.194	11.202	23.721	49.292
	T_{Q2}	4.253	11.312	23.947	49.633
	$\bar{T}$	4.326	11.338	23.975	49.703
	T_{Q3}	4.360	11.449	24.218	50.044
	σ_T	0.209	0.233	0.391	0.709
Double					
MSPSort	T_{Q1}	3.851	10.521	22.908	48.058
	T_{Q2}	3.917	10.595	23.048	48.684
	$\bar{T}$	4.013	10.725	23.050	49.038
	T_{Q3}	4.110	10.693	23.187	50.166
	σ_T	0.222	0.422	0.202	1.151
BQSort	T_{Q1}	3.937	10.754	23.399	49.553
	T_{Q2}	4.093	10.962	23.711	50.771

(Continued)

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Table 6 Continued

Alg.	KPI (Sec.)	200M	500M	1000M	2000M
MWSort	$\bar{T}$	4.197	11.235	23.769	50.883
	T_{Q3}	4.413	11.283	24.183	52.384
	σ_T	0.266	0.706	0.458	1.484
	T_{Q1}	4.247	11.361	24.243	50.250
	T_{Q2}	4.522	11.873	25.080	51.807
MWSort	$\bar{T}$	4.522	11.857	25.122	52.190
	T_{Q3}	4.696	12.213	25.966	54.108
	σ_T	0.312	0.607	0.939	2.212

Table 7 Statistics of Run Time T of MSPSort vs BQSort vs MWSort for all data types at various sizes N on R7-1700 system after 1000 trials

Alg.	KPI (Sec.)	200M	500M	1000M	2000M
UInt32					
MSPSort	T_{Q1}	1.722	4.416	9.382	19.238
	T_{Q2}	1.735	4.438	9.413	19.294
	$\bar{T}$	1.746	4.476	9.418	19.307
	T_{Q3}	1.773	4.561	9.445	19.353
	σ_T	0.032	0.083	0.063	0.120
BQSort	T_{Q1}	1.780	4.643	9.897	20.358
	T_{Q2}	1.811	4.722	10.051	20.695
	$\bar{T}$	1.807	4.725	10.026	20.635
	T_{Q3}	1.827	4.778	10.125	20.845
	σ_T	0.032	0.101	0.160	0.316
MWSort	T_{Q1}	1.973	5.096	10.436	21.290
	T_{Q2}	2.145	5.470	11.114	22.498
	$\bar{T}$	2.109	5.389	10.959	22.214
	T_{Q3}	2.187	5.549	11.241	22.696
	σ_T	0.041	0.086	0.146	0.236
UInt64					
MSPSort	T_{Q1}	2.149	5.723	12.046	25.406
	T_{Q2}	2.161	5.744	12.091	25.494
	$\bar{T}$	2.163	5.752	12.104	25.514
	T_{Q3}	2.174	5.772	12.144	25.584
	σ_T	0.022	0.048	0.092	0.168
BQSort	T_{Q1}	2.137	5.706	11.990	25.231
	T_{Q2}	2.153	5.746	12.077	25.423
	$\bar{T}$	2.160	5.762	12.102	25.487
	T_{Q3}	2.177	5.805	12.196	25.671
	σ_T	0.033	0.083	0.158	0.348
MWSort	T_{Q1}	2.216	5.845	12.184	25.469
	T_{Q2}	2.225	5.864	12.223	25.625
	$\bar{T}$	2.227	5.868	12.228	26.223
	T_{Q3}	2.236	5.887	12.268	27.148

(Continued)

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Table 7 Continued

Alg.	KPI (Sec.)	200M	500M	1000M	2000M
	σ_T	0.015	0.033	0.063	0.890
Double					
MSPSort	T_{Q1}	2.312	6.094	12.699	26.616
	T_{Q2}	2.324	6.120	12.749	26.720
	$\bar{T}$	2.327	6.125	12.757	26.745
	T_{Q3}	2.338	6.146	12.805	26.829
	σ_T	0.026	0.046	0.090	0.210
BQSort	T_{Q1}	2.312	6.097	12.721	26.568
	T_{Q2}	2.327	6.134	12.799	26.751
	$\bar{T}$	2.333	6.147	12.830	26.810
	T_{Q3}	2.347	6.188	12.906	26.942
	σ_T	0.030	0.074	0.159	0.631
MWSort	T_{Q1}	2.735	7.037	14.366	29.394
	T_{Q2}	2.774	7.106	14.485	29.608
	$\bar{T}$	2.778	7.121	14.505	29.628
	T_{Q3}	2.818	7.196	14.626	29.838
	σ_T	0.051	0.120	0.191	0.333

Table 8 Statistics of Run Time T of MSPSort vs BQSort vs MWSort for all data types at various sizes N on X5670 system after 1000 trials

Alg.	KPI (Sec.)	200M	500M	1000M	2000M
UInt32					
MSPSort	T_{Q1}	1.587	4.139	8.334	16.708
	T_{Q2}	1.601	4.177	8.408	16.845
	$\bar{T}$	1.605	4.184	8.440	16.907
	T_{Q3}	1.618	4.216	8.500	17.000
	σ_T	0.027	0.072	0.171	0.334
BQSort	T_{Q1}	1.684	4.039	8.145	16.576
	T_{Q2}	1.692	4.057	8.176	16.662
	$\bar{T}$	1.691	4.073	8.215	16.757
	T_{Q3}	1.699	4.088	8.209	16.788
	σ_T	0.011	0.063	0.155	0.304
MWSort	T_{Q1}	1.686	4.039	8.155	16.642
	T_{Q2}	1.693	4.055	8.183	16.708
	$\bar{T}$	1.692	4.070	8.235	16.819
	T_{Q3}	1.699	4.078	8.223	16.840
	σ_T	0.011	0.062	0.168	0.300
UInt64					
MSPSort	T_{Q1}	2.696	6.694	13.360	24.210
	T_{Q2}	2.736	6.829	13.663	24.831
	$\bar{T}$	2.746	6.843	13.688	25.004
	T_{Q3}	2.788	6.980	14.024	25.582

(Continued)

Table 8 Continued

Alg.	KPI (Sec.)	200M	500M	1000M	2000M
	σ_T	0.072	0.206	0.466	1.065
BQSort	T_{Q1}	2.543	6.340	12.695	24.110
	T_{Q2}	2.584	6.344	12.951	24.953
	$\bar{T}$	2.601	6.417	13.021	25.315
	T_{Q3}	2.638	6.525	13.270	26.085
	σ_T	0.086	0.275	0.491	1.767
MWSort	T_{Q1}	2.065	5.103	10.166	NA
	T_{Q2}	2.085	5.133	10.753	NA
	$\bar{T}$	2.078	5.170	10.693	NA
	T_{Q3}	2.100	5.205	10.880	NA
	σ_T	0.035	0.113	0.492	NA
Double MSPSort	T_{Q1}	2.737	6.672	13.497	24.569
	T_{Q2}	2.771	6.774	13.808	25.186
	$\bar{T}$	2.780	6.796	13.835	25.334
	T_{Q3}	2.812	6.892	14.139	25.808
	σ_T	0.068	0.186	0.468	1.065
BQSort	T_{Q1}	2.610	6.414	13.032	24.890
	T_{Q2}	2.647	6.495	13.248	25.601
	$\bar{T}$	2.664	6.534	13.321	25.981
	T_{Q3}	2.697	6.603	13.547	26.725
	σ_T	0.078	0.187	0.441	1.650
MWSort	T_{Q1}	2.245	5.711	11.717	NA
	T_{Q2}	2.262	5.769	11.787	NA
	$\bar{T}$	2.258	5.724	11.790	NA
	T_{Q3}	2.277	5.811	11.878	NA
	σ_T	0.031	0.149	0.197	NA

Table 9 Statistics of Run Time T of MSPSort vs BQSort vs MWSort for all data types at various sizes N on R9-2920 system after 1000 trials

Alg.	KPI (Sec.)	200M	500M	1000M	2000M
UInt32					
MSPSort	T_{Q1}	1.171	2.972	6.124	12.589
	T_{Q2}	1.181	2.991	6.157	12.659
	$\bar{T}$	1.182	2.994	6.173	12.681
	T_{Q3}	1.191	3.012	6.203	12.730
	σ_T	0.017	0.033	0.086	0.158
BQSort	T_{Q1}	1.237	3.221	6.727	13.971
	T_{Q2}	1.261	3.287	6.859	14.375
	$\bar{T}$	1.264	3.303	6.891	14.488
	T_{Q3}	1.285	3.352	6.991	14.831
	σ_T	0.040	0.127	0.288	0.774
MWSort	T_{Q1}	1.237	3.125	6.423	14.343

(Continued)

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Table 9 Continued

Alg.	KPI (Sec.)	200M	500M	1000M	2000M
	T_{Q2}	1.248	3.142	6.850	14.524
	$\bar{T}$	1.260	3.175	6.774	14.454
	T_{Q3}	1.269	3.197	6.942	14.686
	σ_T	0.035	0.075	0.294	0.367
Uint64					
MSPSort	T_{Q1}	1.680	4.514	9.602	20.180
	T_{Q2}	1.691	4.547	9.678	20.353
	$\bar{T}$	1.694	4.556	9.690	20.357
	T_{Q3}	1.703	4.588	9.771	20.537
	σ_T	0.023	0.065	0.148	0.330
BQSort	T_{Q1}	1.703	4.549	9.529	20.332
	T_{Q2}	1.732	4.638	9.742	20.815
	$\bar{T}$	1.746	4.682	9.838	20.980
	T_{Q3}	1.775	4.769	10.048	21.448
	σ_T	0.063	0.205	0.483	0.990
MWSort	T_{Q1}	1.457	3.898	8.106	16.388
	T_{Q2}	1.474	3.997	8.207	16.584
	$\bar{T}$	1.474	3.946	8.190	16.582
	T_{Q3}	1.489	4.050	8.300	16.766
	σ_T	0.027	0.159	0.187	0.356
Double					
MSPSort	T_{Q1}	1.747	4.679	9.757	20.611
	T_{Q2}	1.759	4.708	9.826	20.756
	$\bar{T}$	1.762	4.718	9.837	20.777
	T_{Q3}	1.772	4.744	9.905	20.906
	σ_T	0.024	0.069	0.118	0.263
BQSort	T_{Q1}	1.756	4.677	9.806	20.655
	T_{Q2}	1.782	4.760	10.003	21.051
	$\bar{T}$	1.798	4.799	10.081	21.306
	T_{Q3}	1.826	4.871	10.253	21.688
	σ_T	0.059	0.173	0.418	1.047
MWSort	T_{Q1}	1.554	3.938	8.791	17.919
	T_{Q2}	1.566	3.960	8.877	18.096
	$\bar{T}$	1.570	4.028	8.732	18.044
	T_{Q3}	1.582	4.002	8.936	18.243
	σ_T	0.024	0.160	0.342	0.406

With respect to MWSort, MWSort was unable to test at $N=2000M$ of Uint64 and Double on X5670 system because the amount of RAM was limited to 24 GB. MWSort can achieve faster average Run Time $\bar{T}$ and low σ_T for all data sizes. It could be due to balanced and independent memory accesses. Both X5670 and R9-2920 systems are NUMA with 4 memory

channels supporting high memory traffic. The tradeoffs between run time and memory resources are still debatable especially on server systems that CPU cores and memory are shared among many processes/threads.

4.6.3 Run Time Stability

It can be noticed that almost all of the run time statistics on every system are right skew where $\bar{T}$ is mostly higher than T_{Q2} (median). For stability analyses, run time statistics σ_T and T_{IQR} can be of interests. The σ_T and T_{IQR} of MSPSort are mostly lower than BQSort and MWSort for every data type except on X5670 system. It can be concluded that MSPSort is consistently stable on a wide variety of systems.

5 Conclusions and Future Work

MSPPartition is a block-based multithreaded version of the single-pivot Hoare's partition algorithm. A number of threads are forked to compare-swap left and right data from both ends to the middle. Each thread has its own private left and right stacks to keep track of those block boundary indices. The partition process continues until the stack on either side is empty first. At last, the sequential Lomuto's is invoked to finish the small leftover region.

The MSPPartition can be recursively applied to become a parallel MSPSort on manycore and even NUMA systems. MSPSort is evaluated on four Linux systems and benchmarked against two STL parallel mode algorithms namely, BQSort and MWSort. MSPSort can achieve better run time statistics than BQSort for all data types and sizes except on Intel X5670 system. However, only MWSort can take advantages of NUMA systems for Uint64 and Double over MSPSort.

As future works, other candidate parameters shall be investigated further to be parameterized as functions of core count. Block size B should be fine-tuned to align with virtual memory page so that cache/TLB misses can be minimized. Different data distributions shall be experimented. In addition, MSPPartition shall be applied to support parallel multipivot partition operations.

References

- [1] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. In-place parallel super scalar samplesort (ipssso). *25th European Symposium on Algorithms: ESA 2017*, 2017.

- [2] Eduard Ayguadé, Nawal Copt, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guan-song Zhang. The design of openmp tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.
- [3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
- [4] Philip Heidelberger, Alan Norton, and John T. Robinson. Parallel quicksort using fetch-and-add. *IEEE Transactions on Computers*, 39(1):847–857, January 1990.
- [5] Basel A. Mahafzah. Performance assessment of multithreaded quicksort algorithm on simultaneous multithreaded architecture. *J of Supercomputing*, 66(1):339–363, 2013.
- [6] Duhu Man, Yasuaki Ito, and Koji Nakano. An efficient parallel sorting compatible with the standard qsort. In *International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 512 – 517, Hiroshima, Japan, December 8-11 2009.
- [7] Duhu Man, Yasuaki Ito, and Koji Nakano. An efficient parallel sorting compatible with the standard qsort. *International Journal of Foundations of Computer Science*, 22(05):1057–1071, 2011.
- [8] David R. Musser. Introspective sorting and selection algorithms. *Software: Practice and Experience*, pages 983–993, 1997.
- [9] Ratthaslip Ranokpanuwat and Surin Kittitornkun. Parallel partition and merge quicksort (ppmqsort) on multicore cpus. *J of Supercomputing*, 72(3):1063–1091, 2016.
- [10] A. Rattanatanurak. Dual parallel partition sorting algorithm. In *Proceedings of 2018 the 8th International Workshop on Computer Science and Engineering, WCSE 2018*, pages 685–690, 2018.
- [11] Johannes Singler, Peter Sanders, and Felix Putze. Mcstl : The multi-core standard template library. *Euro-Par 2007 Parallel Processing. Springer Berlin Heidelberg*, pages 682–694, 2007.
- [12] Michael Süß and Claudia Leopold. A user’s experience with parallel sorting and openmp. In *Proceedings of the Sixth European Workshop on OpenMP-EWOMP 2004*, pages 23–38, 2004.
- [13] Daouda Traoré, Jean-Louis Roch, Nicolas Maillard, and Thierry Gautier. Deque-free work-optimal parallel stl algorithms. *Euro-Par 2008–Parallel Processing. Springer Berlin Heidelberg*, pages 887–897, 2008.

316 A. Rattanatanurak and S. Kittitornkun

- [14] Philippas Tsigas and Yi Zhang. A simple, fast parallel implementation of quicksort and its performance evaluation on sun enterprise 10000. In *11th Euromicro Conference on Parallel Distributed and Network based Processing (PDP 2003)*, pages 372–381, Genoa, Italy, February 5th-7th 2003.
- [15] John Valois. Introspective sorting and selection revisited. *Software: Practice and Experience*, pages 617–638, 2000.

Biographies



Apisit Rattanatanurak received his M.Eng. and B.Eng. degrees in Computer Engineering from King Mongkut's Institute of Technology Ladkrabang (KMITL), Bangkok, Thailand. Now, he is pursuing a doctoral degree at the Faculty of Engineering, KMITL. His research interest is in the area of parallel programming, computing on multi-core CPU and GPU on Linux/Unix system.



Surin Kittitornkun received his Ph.D. and M.S. degrees in Computer Engineering from University of Wisconsin-Madison, USA. Currently, he is an Assistant Professor at Faculty of King Mongkut's Institute of Technology Ladkrabang (KMITL), Bangkok, Thailand. His research interests include parallel algorithms, mobile/high performance computing and computer architecture.

Optimizing MultiStack Parallel (MSP) Sorting Algorithm

Apisit Rattanatanurak and Surin Kittitornkun

*Dept. of Computer Engineering, School of Engineering,
King Mongkut's Institute of Technology Ladkrabang, Bangkok, Thailand 10520*

Abstract

Mobile smartphones/laptops are becoming much more powerful in terms of core count and memory capacity. Demanding games and parallel applications/algorithms can hopefully take advantages of the hardware. Our parallel MSPSort algorithm is one of those examples. However, MSPSort can be optimized and fine tuned even further to achieve its highest capabilities. To evaluate the effectiveness of MSPSort, two Linux systems are quad core ARM [Cortex-A72](#) and 24-core AMD ThreadRipper R9-2920. It has been demonstrated that MSPSort is comparable to the well-known parallel standard template library sorting functions, i.e. Balanced QuickSort and Multiway MergeSort in the various aspects such as run time and memory requirements.

Keywords: Partition, Sort, Multithread, OpenMP, Optimize, Synchronization.

1 Introduction

Mobile manycore CPUs are pervasive in both smartphones and tablet computers. Smartphones' CPUs are becoming multicore towards manycore to support a variety of multimedia applications. Therefore, basic computing algorithms shall be adapted to exploit that. Parallel sorting and data partitioning algorithms [4, 5, 7, 8] are based on the well known single-pivot Hoare's

River Journal, 1–23.

© 2021 River Publishers. All rights reserved.

2 A. Rattanatrakun and S. Kittitornkun

algorithm [1]. It is also known as divide and conquer (D&Q) behavior of QuickSort.

Those algorithms can be broken into two phases: first level partition and partition & sort phases. The first phase is parallel compare and swap (C&S) between elements followed by a cleanup algorithm and sequential partition. The second phase recursively partitions two long subarrays and finally sorts the short ones in sequential.

The first level partition is the bottleneck of D&Q Hoare's algorithm. This paper intends to tackle this problem with simple multithreading techniques while minimizing the unnecessary memory accesses. Based on our Multi-Stack Parallel Sorting (MSPSort) [6], we optimize and fine tune it further. Unlike other F&A parallel block-based QuickSort algorithms, MSPSort enforces data parallelism with dual stacks of data blocks private to each thread. In addition, MSPSort is based on barrier synchronization to handle the classic pivot skew problem. However, the cleanup algorithm can be improved even more.

Our key contributions are as follows. First, the MSPPartition can be adaptive with respect to pivot value. It can adapt to the pivot skew such that fewer unnecessary recursive calls can be reduced while cleaning up the first level partition phase. Second, dynamic thread allocation can increase CPU cores to handle much longer subarray due to pivot skew. This can be beneficial for huge workloads on manycore CPUs. Third, the number of partitioning threads or CPU cores can be adjusted to support a wide variety of data types. As a result, fewer memory accesses are required instantaneously thus lower pressure to the memory hierarchy. Workloads can be shared among partition threads where more free cores are available to sort subarrays in parallel. Our optimization methods are considered global because they can be effectively applied to any manycore CPU system.

We shall present the paper in the following order. Section 2 reviews related background and previous work of parallel D&Q sorting algorithms. The optimized and fine tuned MSPPartition and MSPSort are elaborated in Section 3. Consecutively, experiment results are compared on AMD R9-2920 and Raspberry Pi4 systems. The last section is Conclusions and Future Work.

2 Background & Related work

This section consists of the following subsections, *Standard Template Library Sequential/Parallel Sorting Functions* and *Anatomy of QuickSort Algorithms*.

2.1 Standard Template Library Sequential/Parallel Sorting Functions

The Standard Template Library (*STL*) *Sort* is a sequential sorting function for any data type. It is available in almost C++ compilers and prototyped as follow.

```
std::sort(RandomAccessIterator first,
          RandomAccessIterator last);
```

On the other end of spectrum, Balanced QuickSort or BQSort as mentioned earlier is a classic parallel/multithreaded quicksort algorithm [7]. Each thread assign blocks of data to compare&swap (C&S). A double ended queue of each thread holds the block boundaries and allows other threads to access from the second end. MultiWay Merge Sort or MWSort equally divides data into several subarrays and *std :: sort* them [7]. Each subarray can be independently sorted in parallel and stored a temporary array. It requires an extra space of memory with complexity of N where N is the data size. It makes use of a parallel multiway merging algorithm to merge the sorted subarrays and returns. As a result, its run time is more stable compared with QuickSort algorithm.

2.2 Anatomy of Parallel QuickSort Algorithms

Parallel QuickSort algorithms have been proposed by [4, 5, 7, 8] and ordered in time series. The recent one is called MSPSort by us [6]. All of them can be decomposed into two phases: first level partition and partition & sort phases. The **First Level Partition Phase** is the most challenging task for parallelization. It consists of parallel compare and swap (C&S) between elements followed by a cleanup algorithm and sequential partition. Due to its comparison based nature and global memory accesses, it is limited by latency of address translation look aside buffer (TLB) and page table all the way to cache misses. It can be regarded as *memory bound* problem. In addition, CPU hardware constraints for some basic data types such as floating-point numbers still exist. The second phase called **Partition&Sort** is to recursively partition two long subarrays while traversing the resulting tree of subarrays in multithread adn to finally sort a shorter one by a single thread. Only major algorithms are summarized and compared with ours as follows.

In 1990, Heidelberg et al. [5] first proposed a block-based parallel partition algorithm on shared memory multiprocessor system. Reserving blocks of data to C&S must be synchronized using F&A instruction. Although it is initially inplace, the cleanup phase needs two extra buffers to keep track

4 A. Rattanatrakun and S. Kittitornkun

of elements' indices to be swapped. The next phase can be accomplished using two global queues. The first queue keeps the longer subarrays to be partitioned in parallel with proportional number of processors. The second queue keeps shorter ones to be partitioned sequentially and consequently sorted. Several scheduling algorithms can be applied. The entire algorithm was simulated rather implemented.

PQuickSort [8] was first implemented a block based parallel partition algorithm on a shared memory multiprocessor system in 2003. Each processor reserves a pair of blocks from both left and right sides to C&S data elements. The process continues to reserve one more fresh block from either on left or right side. All processors stop until no fresh block is available. Some unfinished blocks must be swapped to separate between the left and right finished blocks. The sequential partition eventually finishes the first-level partition. The next step assigns a number of processors proportional to subarrays' sizes. Special stack is instantiated to store subarrays and shared among processors to partition and finally sorted them.

MCSTL BQSort [7] as mentioned earlier is also block based and recursive C&S. Fetch and add (F&A) synchronization is needed every time a data block is reserved. All threads stop when their C&S indices get closer within a block distance. Some data swaps must be carried out to prepare for the next recursive function call. Eventually, the sequential partition finishes the first level partition phase. Next, each thread's local deque (double ended queue) can keep track of subarrays to be partitioned with shorter partition first (SPF) traversal. As a result, an idle thread can steal workload from the other end of a non-empty deque.

A year later, Frias and Petit [4] used the same F&A synchronization scheme. However, they propose a new cleanup algorithm to avoid extra C&S. They apply a complete binary tree structure to keep track of the unfinished blocks so that they can be shared to balance workload of all CPU cores. Left data elements shall be swapped only with the right ones according to the leaves and finish up with sequential partition.

The first paper implementing QuickSort on GPU was published by [2] in 2010. Although it is not in-place like others, it consists of two phases. Phase I, the input array is divided into sections to fit GPU local memory. Each section is assigned to a block of threads to C&S within a section. The resulting two subsections will be copied to a buffer with synchronizations due to large number of threads. Phase II, recursive partition continues and uses stack to keep track of the shortest subarrays. Subarrays are partitioned and sorted given a threshold.

Our MSPPartition/Sort [6] is a simple yet efficient block based C&S algorithm. Each thread has its own private left and right stacks of data blocks to enforce data parallelism without any F&A instruction. Only a barrier synchronization is needed when either left or right stack runs out for all threads. As a cleanup algorithm, recursive MSPPartition or a sequential one can be invoked while sacrificing C&S some elements again thanks to current smart branch prediction. In the next phase, combinations of breadth first and depth first traversals can partition to exploit cache locality and consequently *std :: sort* them. No work stealing is necessary.

3 Optimized MSPSort Algorithm

Various factors of our MSPPartition can affect its performance such as block size, how to handle the leftover data, how to balance the workload between left and right partitions, etc.

MSPPartition and MSPSort can be optimized and fine-tuned in the following aspects: block size, pivot value skew handling, number of threads in the first level, middle levels and final level of recursion. We can review and improve the MSPartition in pseudocode as shown in Alg. 1.

3.1 Block Size B

This *MultiStack Parallel Partition* function [6] The Recursive MultiStack Parallel Partition Phase can be decomposed of 2 steps: Parallel Stacked Blocks Partition Step and Unfinished Blocks Partition Step. The *Parallel Stacked Blocks Partition Step* divides an unsorted (sub)array $A = A[0], A[1], \dots, A[N - 1]$, into left and right sides according to the pivot index p . Each side is again divided into blocks of $b = B/\text{sizeof}(\text{Type})$ elements from both ends (Alg. 1, line 4) where B is a block size in KB (Kilo Bytes). Data blocks from left and right sides are assigned to τ threads in round robin to enforce data parallelism and avoid data repetition. Within its own assigned data blocks, each thread can C&S data just like Hoare's algorithm. Therefore, there is no data race among threads.

3.2 Handling Pivot Value Skew in MSPPartition

To ideally balance workload and achieve maximum parallelism, each thread is initially assigned with about the same number of blocks. Each thread has its own private variables (local to each thread) i and j that are left and right

Algorithm 1: Optimized Multi-Stack Partition Algorithm

```

1 Function MSPPartition( $A, i_L, j_R, p, \tau$ )
2    $N_L \leftarrow (p - i_L)/b$  // Number of blocks on left side
3    $N_R \leftarrow (j_R - p)/b$  // Number of blocks on right side
4   for  $i \leftarrow 0$  to  $N_L - 1, i++$  do
5      $L_s[i \bmod \tau].push(i_L + i, i_L + i + b)$  // Push left block indices
6   end
7   for  $i \leftarrow 0$  to  $N_R - 1, i++$  do
8      $R_s[i \bmod \tau].push(j_R - i - b, j_R - i)$  // Push right indices
9   end
10  OpenMP parallel for private( $i, j, l_b, r_b$ )
11  for  $t \leftarrow 0$  to  $\tau - 1$  do
12    while  $!L_s[t].empty() \ \&\& \ !R_s[t].empty()$  do
13       $(i, l_b) \leftarrow L_s[t].pop()$  // Pop left top indices
14       $(r_b, j) \leftarrow R_s[t].pop()$  // Pop right top indices
15    do
16      while  $A[i] \leq A[p] \ \&\& \ i \leq l_b$  do
17         $i++$  // Increase i index
18      end
19      while  $A[j] > A[p] \ \&\& \ j \geq r_b$  do
20         $j--$  // Decrease j index
21      end
22      if  $i \leq l_b \ \&\& \ j \geq r_b$  then
23         $SWAP(A[i++], A[j--])$  // Swap and continue
24      end
25      while  $i \leq l_b \ \&\& \ j \geq r_b$ 
26        if  $i > l_b$  then
27           $R_s[t].push(r_b, j)$  // Push the right indices
28        end
29        else if  $j < r_b$  then
30           $L_s[t].push(i, l_b)$  // Push the left indices
31        end
32      end
33    end
34     $l_{min} \leftarrow \min(L_s[t], \forall t)$  // Barrier Sync to get the left most
      index
35     $r_{max} \leftarrow \max(R_s[t], \forall t)$  // Get the right most index
36     $\mu \leftarrow \text{GetMu}(i_L, j_R, r_{max}, l_{min}, u_{sort}, \tau)$  // get mu threads
37     $\pi \leftarrow \text{GetIndex}(i_L, j_R, r_{max}, l_{min}, p)$  // Find the new pivot index
38     $SWAP(A[p], A[\pi])$  // Swap to new pivot index
39    if  $\mu > 1$  then
40      return  $MSPPartition(A, l_{min}, r_{max}, \pi, \mu)$  // Recursion again
41    end
42    else
43      return  $\text{LomutoPartition}(A, l_{min}, r_{max}, \pi)$  // Sequential
        Partition
44    end
45 EndFunction

```

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

indices within the current left and right blocks, respectively. In addition, two private variables, l_b and r_b are the boundaries of the current left and right blocks, respectively. The data from both sides are C&S within each iteration until either left or right index reaches its boundary (Alg. 1, lines 15 to 24). Next, the current index pair either (i, l_b) or (r_b, j) of the latest unfinished block is push back to its corresponding stack (Alg. 1, lines 27 and 30). In the next iteration, the boundaries of a new block and the unfinished one will be popped off to continue. This loop of each thread stops when either its left or right stack is empty (Alg. 1, lines 12).

After that, $l_{min} = \min(L_s[t], \forall t)$ and $r_{max} = \max(R_s[t], \forall t)$ of all τ threads, must be determined. The number of threads $\mu = \text{GetMu}()$ function (Alg. 1, line 36) can be obtained based on how many folds $r_{max} - l_{min}$ longer than u_{sort} and the current number of threads τ in Eq. (1),

$$\mu = \begin{cases} \tau & , \mu' \geq \tau \\ \mu' & , \mu' < \tau \end{cases} \quad (1)$$

where

$$\mu' = \lfloor \frac{r_{max} - l_{min} + 1}{u_{sort}} \rfloor \quad (2)$$

Next, the *Unfinished Blocks Partition Step* must take care of the pivot value skew issue. This step is the most critical part due to the random distribution of data and pivot value selection of *MO5()*. Earlier [6], the number of blocks on both sides must equal such that $\pi = (r_{max} + l_{min})/2$. However, pivot value skew can lead to the time consuming recursive calls to eventually partition the unfinished part. In this optimized MSPSort, the target index π can be adaptive to the pivot value skew such that the number of recursions can be optimized. Therefore, it should be proportional to the finished (swapped) data elements as shown in Eq. (3).

$$\pi = i_L + \frac{(l_{min} - i_L) \times (r_{max} - l_{min})}{(l_{min} - i_L + j_R - r_{max})} \quad (3)$$

where $l_{min} - i_L$ and $j_R - r_{max}$ represent the lengths of left and right already swapped data, respectively and $r_{max} - l_{min}$ is the length of unfinished data elements.

In summary, the long unfinished part due to extreme pivot value skew can specify the number of μ threads of **MSPPartition()** and estimate the next pivot index π from Eq.(3) (Alg. 1, line 40) in the *Unfinished Blocks*

8 A. Rattanatrakun and S. Kittitornkun

Partition Step. Otherwise, the Lomuto's Partition [3] sequentially returns the pivot index (Alg. 1, line 43).

3.3 Partition Thread Ratio θ

Based on our observations in the previous work [6], faster partition time can be also achieved at $\tau < \tau_{max}$ as well. As a result, the initial number of threads to partition can be parameterized to $\theta \times \tau_{max}$ where θ is the Partition Thread Ratio. At line 2 of Alg. 2, if $\theta < 1.00$, it shall relieve some pressure of the last level caches as well as the data TLBs where some partitioning threads must access data (memory) at various locations simultaneously.

Algorithm 2: Modified MSPSort Algorithm

```

1 Function Main()
2   MSPSort( $A, 0, N - 1, \theta \tau_{max}$ )    // MSPSort with  $\theta \times \tau_{max}$  threads
3 End
4 Function MSPSort( $A, i_L, j_R, \tau$ )
5    $p \leftarrow MO5(A, i_L, j_R)$           //  $p$ =Median of Five
6   if  $j_R - i_L > u_{sort}$  then
7      $p \leftarrow MSPPartition(A, i_L, j_R, p, \tau)$     // with  $\tau$  threads
8     if  $\tau / \alpha > \tau_{min}$  then
9        $\tau \leftarrow \tau / \alpha$           // Reduce  $\tau$  threads by  $\alpha$ 
10    end
11  else
12     $\tau \leftarrow \tau_{min}$           // Saturate at  $\tau_{min}$  threads
13  end
14  if  $j_R - i_L > u_{trav}$  then
15    BF-MSPSort( $A, i_L, j_R, p, \tau$ )    // Breadth First  $\tau$  threads
16  end
17  else
18    DF-MSPSort( $A, i_L, j_R, p, \tau$ )    // Depth First with  $\tau$  threads
19  end
20 end
21 else
22   OpenMP Task untied
23    $std :: sort(A + i_L, A + j_R)$     // Fork an untied thread
24 end
25 End

```

3.4 Dynamic Thread Allocation

The Dynamic Thread Allocation consists of two independent parameters: Thread Divider α and Thread Equalizer β . The Thread Divider $1 < \alpha < 3$ is between each recursive level. β is the Thread Equalizer of the next recursion level to compensate with pivot skew from the previous recursion. That means longer subarray on either left or right side may require more threads than its initial allocation. If Thread Equalizer $\beta=ON$, each task will get different number of threads, τ_L and τ_R , respectively. Otherwise, $\beta=OFF$, $\tau_L=\tau_R=\tau$ in Equations (4) and (5).

$$\beta_L = \begin{cases} 1.5 & , l/r \geq 3 \\ 1 + \frac{l-r}{l+r} & , 1 \leq l/r < 3 \\ 1 & , otherwise \end{cases} \quad (4)$$

and

$$\beta_R = \begin{cases} 1.5 & , r/l \geq 3 \\ 1 + \frac{r-l}{l+r} & , 1 \leq r/l < 3 \\ 1 & , otherwise \end{cases} \quad (5)$$

3.5 Depth First Traversal Algorithms

In the **MSPSort()** function, Median of Five function **MO5()** (Alg. 2, line 5) selects a pivot index p and swaps it to the middle of subarray A . The **MSPPartition()** partitions the subarray A with respect to the pivot index p and finally returns it back (Alg. 2, line 7). **MSPSort()** continues traversing the tree of subarrays with Breadth-First (Alg. 2, line 15) by default. The details of **BF-MSPSort()** can be explored in Alg. 3. It can be noticed that **MSPSort()** is invoked as two recursive untied OpenMP tasks (Alg. 3, lines 3 and 7) corresponding to the left and right subarrays, respectively.

With the default BF traversal only, any resulting subarray shorter than u_{sort} subarray is sorted as an independent thread (Task) (Alg. 2, line 23) using `std::sort` where $u_{sort} = U_{sort}/sizeof(Type)$, U_{sort} is Sorting Cutoff parameter, and $Type$ corresponds to the data type to be sorted.

To achieve as high parallelism as possible, BF traversal is of the choice. Therefore, its unpredictable execution order can be due to various factors such as the subarray sizes, dTLB misses/page faults and branch/memory stalls. However, cache locality can be exploited at almost the deepest level of recursion. We have been implementing DF (Depth First) traversal algo-

Algorithm 3: Breadth-First and Depth-First MSPSort

```

1 Function BF-MSPSort( $A, i_L, j_R, p, \tau$ )
2    $\tau_L \leftarrow \beta_L \times \tau$  //  $\tau_L$  threads
3   OpenMP Task untied
4   MSPSort( $A, i_L, p-1, \tau_L$ ) // left subarray  $\tau_L$  threads
5    $\tau_R \leftarrow \beta_R \times \tau$  //  $\tau_R$  threads
6   OpenMP Task untied
7   MSPSort( $A, p+1, j_R, \tau_R$ ) // right subarray  $\tau_R$  threads
8 End
9 Function DF-MSPSort( $A, i_L, j_R, p, \tau$ )
10   $P_s.push(i_L, j_R)$  // Push the partition's boundary
11  while ! $P_s.empty()$  do
12     $i_L, j_R \leftarrow P_s.pop()$  // Pop the partition's boundary
13    if  $j_R - i_L > u_{sort}$  then
14       $p \leftarrow MO5(A, i_L, j_R)$  //  $p$ =Median of Five
15       $p \leftarrow MSPPartition(A, i_L, j_R, p, \tau)$  // with  $\tau$  threads
16       $P_s.push(i_L, p-1)$  // Decide to push which partition first
17       $P_s.push(p+1, j_R)$  // Push left side first (RAL)
18      // Push right side later (RAL)
19    end if
20    else
21      OpenMP Task untied
22       $std::sort(A + i_L, A + j_R)$  // Fork an untied thread
23    end else
24  end while
25 End

```

rithm in **DF-MSPSort()** function. Once the subarray $(j_R - i_L)$ is shorter than u_{trav} elements (Alg. 2, line 18), MSPSort switches to DF traversal where $u_{trav} = U_{trav}/sizeof(Type)$ elements and U_{trav} is Traversal Cutoff in MB. The pseudocode in Alg.3 illustrates RAL (Right Always) algorithm as an example.

There are two different approaches of BF-DF traversal, direction versus subarray size oriented. Both RAL and LAL (Left Always) are direction oriented. On the contrary, Longer Partition First (LPF) and Shorter Partition First (SPF) are size oriented. Several hybrid algorithms can be devised.

3.6 Minimum Threads τ_{min}

As the number of subarrays to be processed grows exponentially, deeper recursive level requires fewer threads. As a result, the number of software

threads τ is reduced to τ/α (Alg. 2, line 12) and kept at $\tau = \tau_{min}$ threads where τ_{min} is the minimum number of threads. In order to share CPU cores with other threads while maximizing parallelism, τ_{min} should be fine tuned as well.

4 Experiments, Results & Discussions

The experiments on two different Linux systems. Experiment parameters are listed and rationalized. Consecutively, the obtained results are elaborated and discussed.

4.1 Experiment Setup

Table 1 Specifications of multicore CPUs, KB: Kilobytes, MB: Megabytes

CPU Name	Cortex A72 Pi4	ThreadRipper R9-2920
Clock (GHz)	1.5	3.50
c (cores)	4	24
RAM	8GB	64GB
Configuration	1×8GB	8×8GB
Technology	DDR4	DDR4
Frequency (MHz)	2133	2400
L1 I-Cache	4×48KB 3Way	12×64KB 4Way
L1 D-Cache	4×32KB 2Way	12×32KB 8Way
L2 Cache	1024KB 16Way	12×512KB 8Way
L3 Cache	-	4×8MB 16Way
64-bit OS	Raspberry Pi	Ubuntu
Version	Beta	18.04 LTS
Kernel	5.4.42-v8+	5.3.0-62-generic
G++	8.3.0	7.5.0
OpenMP	4.5	4.5

The fine-tuned MSPSort algorithm can be experimented on two totally different systems as listed in Table 1. They both run the same and G++ version. The number of cores c is reported by Linux System Monitor. Moreover, these systems widely differ in terms of memory size, technology and configuration. Nonetheless, to compare between non-L3 cache CPU and L3-cache CPU cores. R9-2920 represents high-end mobile systems with high CPU core count, large RAM capacity and shared L3 caches. On the other

hand, Raspberry Pi4 represents ordinary mobile systems with low core count, small RAM capacity and a shared L2 cache instead.

Table 2 Experiment parameters of *MSPSort*, BF: Bread-First, DF: Depth-First, $M=10^6$

Parameters	Values
Algorithms	<i>MSPSort</i> , <i>BQSort</i> , <i>MWSort</i>
Data Types	Uint32, Double
Random Dist	Uniform
GCC Optimization	O2
Data size N (Pi4)	20M, 50M, 100M, 200M
Data size N (R9)	200M, 500M, 1000M, 2000M
Traversal	RAL, LAL, LPF, SPF
Block size B	32, 64, 128, 256 KB
Sorting Cutoff U_{sort}	2, 4, 8, 16 MB
Traversal Cutoff U_{trav}	8, 16, 32, 64 MB
Multiplier m	1, 2
Minimum Thread τ_{min}	2, 3
Partition Thread Ratio θ	0.67, 0.75, 0.8, 0.9
Thread Divider α	4/3, 3/2, 2, 5/2

As listed in Table 2, the parameters on both Pi4 and R9-2920 systems are the same except N . The data types to be evaluated include Unsigned 32-bit integer (Uint32) and 64-bit double precision floating point numbers (Double). Only uniform distribution is evaluated. Each algorithm is compiled with -O2 flag to save experiment time. There are two sets of data size N . N ranges from 20M to 200M elements due to system RAM limit for Pi4 and 200M to 2000M for R9-2920.

As mentioned earlier, the block size B , Sorting Cutoff U_{sort} and Traversal Cutoff U_{trav} are **not** parameterized with L3 Cache size 8MB. The block size B is in order of KBs. Sorting Cutoff U_{sort} and Traversal Cutoff U_{trav} are in order of MBs as listed in Table 2. *MSPSort* can fork as many $\tau_{max} = m \times c$ threads where the Multiplier $m = 1, 2$. The minimum threads τ_{min} can be independent of hardware configuration, $\tau_{min} = 2, 3$.

4.2 Key Performance Indicators (KPIs)

In this paper, some experiment results shall be normalized and compared based on these KPIs. They all represent time domain aspects of each sorting algorithm. Although lower $\bar{T}$ and σ_T are better in terms of run time and

stability, other statistics play important roles as well. We shall discuss the experiment results with respect to the following aspects.

4.2.1 Average Run Time ($\bar{T}$) and Run Time per zM ($\bar{T}_z$)

An `omp_get_wtime()` can be invoked twice just before and after running the algorithm to get accurate time stamps of each algorithm. The obtained time difference is the Run Time T . Note that data array randomization and initialization as well as verification after are excluded.

The Average Run Time ($\bar{T}$) can be normalized at z Million elements and called Run Time per zM ($\bar{T}_z$). Based on this, It is easy to compare at any data size for certain experiments. Furthermore, this normalized $\bar{T}_z$ can be system independent and compared among several systems running the same algorithm.

4.2.2 Run Time Standard Deviation σ_T and σ_z

Run Time Standard of Deviation (σ_T) represents the stability of each algorithm due to the randomness of each workload. In addition, the normalized standard deviation (σ_z) per z Million elements can justify some parameters specially Block size B .

4.2.3 Run Time Stability

In addition to arithmetic mean $\bar{T}$ and standard deviation σ_T , the first, second and third quartiles are T_{Q1} , T_{Q2} and T_{Q3} , respectively. The InterQuartile Range can be determined as $T_{IQR} = T_{Q3} - T_{Q1}$ for stability analyses. These statistics can specify how the Run Time T distributes over 1000 trials.

4.2.4 User-Mode CPU Utilization: V_{usr}

The amount of time that the CPU cores spending of a program can be measured in a virtual file `/proc/stat`. The CPU time consists of four major components: user, system, nice and idle modes. We are interested in user mode CPU utilization because the higher user-mode utilization V , the more CPU is occupied by an algorithm. Other tasks shall be slowed down.

4.2.5 RAM Utilization: V_{ram}

RAM can be limited due to cost constraints especially mobile platforms. The MaxResident is the maximum amount memory residing in RAM measured by `/usr/bin/time` Linux command in KiloBytes. Its [man page](#) is available online. Hence, the RAM Utilization percentage can be computed as follow.

$$V_{ram}(\%) = \frac{1024 \times MaxResident \times 100\%}{N \times sizeof(type)} \quad (6)$$

The more memory (RAM) consumption compared to the workload size, the higher V_{ram} percentage.

4.3 Results & Discussions

In order to optimize and fine tune parameters of MSPSort effectively, we start with the bottom most parameters all the way up to the top ones.

4.3.1 BF-MSPSort vs Block Size B

This subsection explains how we fine tune the block size B for Uint32 and Double. Earlier [6] the block size $B=0.01 \times 8\text{MB}$ which is about 80 KB. We start tuning B to be multiples of 32 KB (L1 data cache) on both systems. It is expected that B of larger data type as Double to be twice as big as that of Uint32 as show in Table 3 and Table 4.

Table 3 $\bar{T}_{10}$ and σ_{10} of BF-MSPSort, Uint32 and Double at various block sizes B on Pi4 system after 100 trials

Type	KPI (Sec.)	20M	50M	100M	200M
Uint32					
$B=32\text{KB}$	$\bar{T}_{10}$	0.6396	0.6615	0.7056	0.7332
	σ_{10}	0.0079	0.0058	0.0139	0.0068
$B=64\text{KB}$	$\bar{T}_{10}$	0.6311	0.6627	0.6948	0.7359
	σ_{10}	0.0112	0.0054	0.0071	0.0138
$B=128\text{KB}$	$\bar{T}_{10}$	0.7270	0.7616	0.7870	0.8334
	σ_{10}	0.0206	0.0139	0.0090	0.0180
Double					
$B=64\text{KB}$	$\bar{T}_{10}$	1.0483	1.0978	1.1359	1.2004
	σ_{10}	0.0250	0.0309	0.0193	0.0217
$B=128\text{KB}$	$\bar{T}_{10}$	1.0714	1.0887	1.1576	1.1986
	σ_{10}	0.0203	0.0207	0.0383	0.0245
$B=256\text{KB}$	$\bar{T}_{10}$	1.0253	1.1047	1.1560	1.2007
	σ_{10}	0.0115	0.0298	0.0379	0.0249

The selected values of B are 64 KB for Uint32 and 128 KB for Double on both systems. The focus is on $N=200\text{M}$ and 2000M for Pi4 and R92920, respectively. It can be noticed that these values yield the optimal KPIs in terms of $\bar{T}_{10}$ and σ_{10} as well as. The same reason for R9-2920 is $\bar{T}_{100}$ and σ_{100} instead.

Table 4 $\bar{T}_{100}$ and σ_{100} of BF-MSPSort, Uint32 and Double at various block sizes B on R9-2920 system after 100 trials

Type	KPI (Sec.)	20M	50M	100M	200M
Uint32					
$B=32\text{KB}$	$\bar{T}_{100}$	0.5762	0.6008	0.6244	0.6609
	σ_{100}	0.05762	0.0172	0.0206	0.0285
$B=64\text{KB}$	$\bar{T}_{100}$	0.5774	0.6018	0.6250	0.6533
	σ_{100}	0.0120	0.0147	0.0210	0.0141
$B=128\text{KB}$	$\bar{T}_{100}$	0.5753	0.5963	0.6247	0.6535
	σ_{100}	0.0081	0.0141	0.0164	0.0156
Double					
$B=64\text{KB}$	$\bar{T}_{100}$	0.8775	0.9503	0.9937	1.0484
	σ_{100}	0.0208	0.249	0.0194	0.0317
$B=128\text{KB}$	$\bar{T}_{100}$	0.8785	0.9473	0.9908	1.0325
	σ_{100}	0.0154	0.224	0.0179	0.0382
$B=256\text{KB}$	$\bar{T}_{100}$	0.8776	0.9481	0.9827	1.0397
	σ_{100}	0.0133	0.205	0.0265	0.0411

4.3.2 BF-MSPSort vs U_{sort}, m, τ_{min}

Given a fixed B from the previous experiments, the next parameters, U_{sort} , m and τ_{min} shall be investigated in BF-MSPSort. At the same time, we can learn how they interact to each other given the same data size N . The top (m and τ_{min}) tuples are listed in Table 5. It can be noticed that the whole run time is a trade off between partition time and sorting time.

Table 5 Top ($B, U_{sort}, m, \tau_{min}$) tuples with BF traversal for all N 's, after 20 Trials

System	Pi4	R9-2920
Uint32	(64kB, 4MB, 1, 2)	(64kB, 4MB, 2, 2)
	(64kB, 4MB, 1, 3)	(64kB, 4MB, 1, 3)
	(64kB, 4MB, 2, 2)	(64kB, 4MB, 1, 2)
Double	(128kB, 8MB, 1, 2)	(128kB, 8MB, 1, 2)
	(128kB, 8MB, 2, 2)	(128kB, 8MB, 1, 3)
	(128kB, 4MB, 1, 2)	(128kB, 8MB, 2, 2)

4.3.3 BF-MSPSort vs BF-DF Algorithms

Given the top parameter tuple from BF-MSPSort listed on Table 5, we shall study how each traversal algorithm differs from one another. Earlier, U_{trav} must be configured to multiples of 8MB-L3 cache. In this optimized version, U_{trav} could be adjusted to be any number or as a function of problem size especially for huge ones. It turns out that MSPSort with RAL BF-DF traversal

yields the best $\bar{T}$ and σ_T for both systems and data types. Therefore, RAL is selected and thus called it RAL-MSPSort. Its final parameters are listed in Table 6.

Table 6 Selected top $(B, U_{sort}, U_{trav}, m, \tau_{min})$ tuples for RAL-MSPSort for all N 's, after 100 Trials

System	Uint32	Double
Pi4		
20M-200M	(64KB, 4MB, 4MB, 1, 2)	(128KB, 8MB, 8MB, 1, 2)
R9-2920		
200M	(64KB, 2MB, 8MB, 2, 2)	(128KB, 4MB, 8MB, 1, 2)
500M	(64KB, 4MB, 16MB, 2, 2)	(128KB, 8MB, 16MB, 1, 2)
1000M	(64KB, 8MB, 32MB, 2, 2)	(128KB, 8MB, 32MB, 1, 2)
2000M	(64KB, 8MB, 32MB, 2, 2)	(128KB, 8MB, 64MB, 1, 2)

RAL-MSPSort can yield the best Run Time statistics among all traversal algorithms, even though the BF execution order is similar to SPF algorithm. For both BF and SPF algorithms, shorter subarrays get partitioned done faster the resulting two subarrays can be enqueued sooner. This may not be good for last level cache on average.

With a single shared L2 cache, Pi4 system can be regarded as one-fourth of R9-2920 system. RAL-MSPSort still can yield better Run Time than BF traversal, especially when $U_{trav} = U_{sort}$ because MSPPartition() continues one more time resulting in two subarrays shorter than u_{sort} .

The cutoff may be different U_{trav} due to the data types and a wide variety of cache configurations especially mobile System on Chips. Some of them do not have L3 caches only L2 caches are available instead. In addition, some CPU cores are not identical such as ARM big.Little technology. Cache coherency may be an issue at the block as well as subarray boundaries. These boundaries can fall into the same cache block being accessed by two threads simultaneously.

4.3.4 RAL-MSPSort vs Partition Thread Ratio θ

In this experiment, we shall explore how the number of threads running MSPPartition() affect the total Run Time T . The maximum threads stays the same at τ_{max} only the initial partition threads is reduced to $\theta \times \tau_{max}$ where $\theta \leq 1.00$. Based on parameters in Table 6, $\bar{T}$ and σ_T of RAL-MSPSort at different θ are listed in Table 7 and 8 for Pi4 and R9-2920 systems, respectively.

We can observe that Pi4 loses some performance on both data types as we apply $\theta < 1.00$. For R9-2920, θ of less than 1.00 mitigates that pressure thus yields faster Run Time. On the other hand, parallelism of Pi4 is limited by CPU cores rather than memory bottleneck.

On the contrary, R9-2920 gains significant performance only for Double at $\theta=0.8$ not Uint32. Two major reasons are behind this phenomenon. Firstly, this is because Double data type is twice as big as Uint32 per data element. That means either L3 cache or RAM is twice as busy with Double data requests given the same amount of time as Uint32 data. Secondly, it may be due to floating-point hardware resource constraints as well. Note that R9-2920 is multithreaded CPU and actually a 12-core 24-thread CPU.

Table 7 $\bar{T}$ and σ_T of MSPSort, RAL for Uint32 and Double at various sizes N and Partition Thread Ratio θ on Pi4 system after 100 trials

Alg.	KPI (Sec.)	20M	50M	100M	200M
Uint32	$\bar{T}$	1.2640	3.2878	6.8597	14.3510
	σ_T	0.0269	0.0357	0.0805	0.1368
	$\theta = 0.75$	1.2945	3.3392	6.8744	14.6111
Double	$\bar{T}$	2.1488	5.4478	11.4136	24.2979
	σ_T	0.0491	0.1079	0.2354	0.5938
	$\theta = 0.75$	2.1637	5.5548	11.5595	24.6450
	σ_T	0.0251	0.0666	0.1432	0.2352

4.3.5 RAL-MSPSort vs Thread Divider α

The earlier MSPSort [6], the Thread Divider α is always 2. The question may arise why τ in the next recursion is always divided by 2. Can other values be better than this? Lower $\alpha < 2$ of $5/3$, $3/2$ and $4/3$ can result in more threads in the next recursion level. On the other hand, higher $\alpha > 2$ of $5/2$ can result in fewer threads. Based on the best parameters from the previous experiment, the effects of α on Pi4 and R9-2920 systems are provided in Table 9 and 10, respectively.

It can be observed that $\alpha=2$ yields the best results for Pi4 system and both types. ON R9-2920 at Double $N=1000M$ and $2000M$, the saddle points at $\alpha=5/3$ can be observed instead of $\alpha=2$. High double workload may get benefit from Dynamic Thread Allocation. The effects of Thread Equalizer β

Table 8 $\bar{T}$ and σ_T of MSPSort, RAL for Uint32 and Double at various sizes N and Partition Thread Ratio θ on R9-2920 system after 100 trials

Alg.	KPI (Sec.)	200M	500M	1000M	2000M
Uint32	$\theta = 1.00$ $\bar{T}$	1.1598	2.9678	6.2311	12.5458
		σ_T	0.0146	0.0245	0.1192
	$\theta = 0.90$ $\bar{T}$	1.1459	2.9757	6.1624	12.7117
		σ_T	0.0245	0.0847	0.2307
	$\theta = 0.80$ $\bar{T}$	1.1396	2.9329	6.2100	12.9028
		σ_T	0.0167	0.0388	0.1980
	$\theta = 0.75$ $\bar{T}$	1.1429	2.9468	6.2087	12.8381
		σ_T	0.0140	0.0524	0.2250
	$\theta = 0.67$ $\bar{T}$	1.1437	2.9518	6.2619	12.9881
		σ_T	0.0223	0.0566	0.3486
	$\theta = 1.00$ $\bar{T}$	1.7408	4.6700	9.6917	20.1682
		σ_T	0.0231	0.1736	0.2836
Double	$\theta = 0.90$ $\bar{T}$	1.7363	4.6454	9.5878	20.3231
		σ_T	0.0189	0.1460	0.2413
	$\theta = 0.80$ $\bar{T}$	1.7118	4.6077	9.5261	20.0296
		σ_T	0.0795	0.2034	0.4592
	$\theta = 0.75$ $\bar{T}$	1.7230	4.6286	9.5329	20.1099
		σ_T	0.0318	0.1658	0.4191
	$\theta = 0.67$ $\bar{T}$	1.7197	4.6421	9.6873	20.4310
		σ_T	0.0384	0.1327	0.3766

shall be presented in the last subsection. However, for simplicity $\alpha=2$ for both data types and all systems.

4.3.6 RAL-MSPSort: Final Results vs. Thread Equalizer β

Once the parameters are finalized ($\beta=OFF$), the RAL-MSPSort is evaluated again at 1000 trials to get more stable results and benchmarked with BQSort and MWSort. Run time statistics of all sorting algorithm of Pi4 and R9-2920 are enumerated in Table 11 and Table 12, respectively. On Pi4 system and $N \leq 100M$, all $\bar{T}$ of both data types of all algorithms are similar. At workload $N=200M$, RAL-MSPSort's T_{Q3} to T_{Q1} are the best in all categories. On R9-2920, RAL-MSPSort performs extremely fast on Uint32 data. However, MWSort is still the fastest of Double data type.

For run time stability analyses of both systems, σ_T and T_{IQR} can be of interests. The σ_T and T_{IQR} of RAL-MSPSort are mostly lower than those

Table 9 $\bar{T}$ and σ_T of MSPSort for Uint32 ($\theta = 1.00$) and Double ($\theta = 1.00$) at various sizes N and α, β on Pi4 system after 100 trials

Alg.	KPI (Sec.)	20M	50M	100M	200M
Uint32					
$\alpha = 4/3$	$\bar{T}$	1.2931	3.3368	6.9511	14.5192
$\beta = \text{OFF}$	σ_T	0.0265	0.0568	0.0909	0.2386
$\alpha = 3/2$	$\bar{T}$	1.2931	3.3321	6.8597	14.4847
$\beta = \text{OFF}$	σ_T	0.0297	0.0638	0.0976	0.2391
$\alpha = 2$	$\bar{T}$	1.2640	3.2878	6.8597	14.3510
$\beta = \text{OFF}$	σ_T	0.0269	0.0357	0.0805	0.1368
$\alpha = 5/2$	$\bar{T}$	1.3303	3.4218	7.0489	14.7332
$\beta = \text{OFF}$	σ_T	0.0723	0.0686	0.1065	0.1512
Double					
$\alpha = 5/3$	$\bar{T}$	2.1535	5.5741	11.5349	24.4927
$\beta = \text{OFF}$	σ_T	0.0464	0.1681	0.1554	0.6259
$\alpha = 2$	$\bar{T}$	2.1488	5.4478	11.4136	24.2979
$\beta = \text{OFF}$	σ_T	0.0491	0.1079	0.2354	0.5938
$\alpha = 5/2$	$\bar{T}$	2.2146	5.6488	11.6705	24.6764
$\beta = \text{OFF}$	σ_T	0.0479	0.1959	0.3204	0.5461

of BQSort and MWSort for both data types. It can be concluded that RAL-MSPSort is consistently stable on both low-end and high-end mobile systems.

From average user mode CPU utilization V_{usr} point of view, RAL-MSPSort incurs lower percentage than BQSort but a bit higher than MWSort. That means operations of stacks to interleave C&S data within blocks can be effective yet similar to the original Hoare's. Unlike BQSort, with work stealing, RAL-MSPSort relies on thread scheduling of OpenMP to balance the CPU Utilization. Therefore, Some CPU cores can be available to other urgent tasks while sorting in parallel.

The bigger workload, the smaller RAM Utilization is for all algorithms. The extra percentage over 100% is considered overhead. This overhead stems from the data structures to handle parallel operations. Given the same problem size N , Double data type incurs less amount of overhead percentage. RAL-MSPSort and BQSort need similar amount of RAM for both Uint32 and Double. However, MWSort requires double the RAM spaces as the /usr/bin/time command reports. It can be due to fact that the MultiWay Parallel Merge function [7] needs an extra buffer with space complexity $O(N)$ as shown $\bar{V}_{ram} > 200\%$ in Tables 11 and 12. In other words, MWSort can benefit from memory configuration of R9-2920 much better than the Hoare-like algorithms such as RAL-MSPSort and BQSort.

Table 10 $\bar{T}$ and σ_T of MSPSort for Uint32 ($\theta = 1.00$) and Double ($\theta = 0.80$) at various sizes N and α, β on R9-2920 system after 100 trials

Alg.	KPI (Sec.)	200M	500M	1000M	2000M
Uint32					
$\alpha = 5/3$	$\bar{T}$	1.1603	2.9786	6.2968	12.7600
$\beta = \text{OFF}$	σ_T	0.0167	0.0388	0.1980	0.5081
$\alpha = 2$	$\bar{T}$	1.1598	2.9678	6.2311	12.5458
$\beta = \text{OFF}$	σ_T	0.0146	0.0245	0.1192	0.2564
$\alpha = 5/2$	$\bar{T}$	1.1740	3.0219	6.4761	13.0133
$\beta = \text{OFF}$	σ_T	0.0245	0.0869	0.3986	0.7027
Double					
$\alpha = 3/2$	$\bar{T}$	1.7356	4.6483	9.6046	20.1066
$\beta = \text{OFF}$	σ_T	0.0557	0.1152	0.2038	0.5079
$\alpha = 5/3$	$\bar{T}$	1.7255	4.6356	9.4302	19.8885
$\beta = \text{OFF}$	σ_T	0.0725	0.1871	0.5151	0.9203
$\alpha = 2$	$\bar{T}$	1.7118	4.6077	9.5261	20.0296
$\beta = \text{OFF}$	σ_T	0.0795	0.2034	0.4592	1.0444
$\alpha = 5/2$	$\bar{T}$	1.7889	4.8657	10.2009	21.4310
$\beta = \text{OFF}$	σ_T	0.0724	0.3059	0.6525	1.7900

The Thread Equalizer β is only switch ON for $N=200M$ and $N=2000M$ on Pi4 and R9-2920, respectively. The experiment results are augmented to the final ones at the right most column. Left and right Thread Equalizers β_L and β_R correspond to Equations (4) and (5), respectively. Due to the off balanced pivot selection, the longer subarray is enhanced with more threads. It can be noticed at $N=200M$ that Pi4 system loses both $\bar{T}$ and σ_T . This can be due to small number of $c=4$ cores. However, R9-2920 system has shown that $\beta=\text{ON}$ can handle huge both Uint32 and Double workload, i.e. $N=2000M$ well. Their Run Time statistics are much better than those of $\beta=\text{OFF}$.

5 Conclusions & Future work

Our fine-tuned MSPSort has been proved that its performance is comparable to the Standard Template Library parallel sorting algorithms, i.e. BQSort and MWSort. The comparison is carried out on two mobile-equivalent Linux systems, i.e. quad core ARM Cortex A72 on Raspberry Pi4 and 24-core AMD ThreadRipper representing low-end and high-end systems. The key performance indicators include run time statistics, user-mode CPU utilization and memory (RAM) utilization.

For further improvements, the current stack implementation shall be investigated and replaced with other C++ standard templates. Our MSPPartition shall be applied to solve other fundamental problems such as parallel median, minimum and maximum value selections. A combination of SPF and RAL traversal algorithms shall be explored to gain even more cache locality.

References

- [1] Charles Antony and Richard Hoare. Quicksort. *ACM*, 4:321, 1962.
- [2] Daniel Cederman and Philippas Tsigas. Gpu-quicksort: A practical quicksort algorithm for graphics processors. *J. Exp. Algorithmics*, 14:4:1.4–4:1.24, January 2010.
- [3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
- [4] Leonor Frias and Jordi Petit. *Parallel Partition Revisited*, volume 5038 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008.
- [5] Philip Heidelberger, Alan Norton, and John T. Robinson. Parallel quicksort using fetch-and-add. *IEEE Transactions on Computers*, 39(1):847–857, January 1990.
- [6] Apisit Rattanatanurak and Surin Kittitornkun. A multistack parallel (msp) partition algorithm applied to sorting. *Journal of Mobile Multimedia*, pages 293–316, 2020.
- [7] Johannes Singler, Peter Sanders, and Felix Putze. Mcstl : The multi-core standard template library. *Euro-Par 2007 Parallel Processing*. Springer Berlin Heidelberg, pages 682–694, 2007.
- [8] Philippas Tsigas and Yi Zhang. A simple, fast parallel implementation of quicksort and its performance evaluation on sun enterprise 10000. In *11th Euromicro Conference on Parallel Distributed and Network based Processing (PDP 2003)*, pages 372–381, Genoa, Italy, February 5th-7th 2003.

Author's name. A short vitae can be included here.

22 A. Rattanatanurak and S. Kittitornkun

Table 11 KPIs of of RAL-MSPSort vs BQSort vs MWSort for Uint32 ($\theta = 1.00$) and Double ($\theta = 1.00$) at various sizes N on Pi4 system after 1000 trials, $B=128\text{KB}$, $U_{\text{sort}}=8\text{MB}$, $m=1$, $\tau_{\min}=2$, $\alpha=2$, $\beta=\text{OFF}$

Alg.	KPI (Sec.)	20M	50M	100M	200M	200M
Uint32 MSPSort	T_{Q3}	1.2949	3.3129	6.9138	14.4948	$\beta=\text{ON}$ 14.6631
	$\bar{T}$	1.2749	3.2883	6.8673	14.4120	14.5390
	T_{Q2}	1.2732	3.2866	6.8607	14.3877	14.4783
	T_{Q1}	1.2539	3.2622	6.8136	14.3117	14.3709
	T_{IQR}	0.0410	0.0507	0.1002	0.1831	0.2922
	σ_T	0.0306	0.0374	0.0875	0.1626	0.2313
	$\bar{V}_{usr}(\%)$	92.1907	95.8898	97.1730	97.7721	97.7858
	$\bar{V}_{ram}(\%)$	103.7916	101.6674	100.8546	100.5402	-
BQSort	T_{Q3}	1.2814	3.4008	7.1837	14.9355	
	$\bar{T}$	1.2682	3.3726	7.1146	14.8235	
	T_{Q2}	1.2640	3.3663	7.0907	14.8122	
	T_{Q1}	1.2482	3.3389	7.0259	14.6850	
	T_{IQR}	0.0332	0.0619	0.1578	0.2505	
	σ_T	0.0327	0.0480	0.1268	0.1986	
	$\bar{V}_{usr}(\%)$	98.2782	98.6422	98.8069	98.9111	
	$\bar{V}_{ram}(\%)$	103.9911	101.5996	100.7971	100.4020	
MWSort	T_{Q3}	1.2517	3.2947	6.9192	14.6352	
	$\bar{T}$	1.2401	3.2739	6.8760	14.5602	
	T_{Q2}	1.2354	3.2678	6.8708	14.5445	
	T_{Q1}	1.2243	3.2469	6.8253	14.4635	
	T_{IQR}	0.0274	0.0478	0.0939	0.1717	
	σ_T	0.0243	0.0400	0.0693	0.1453	
	$\bar{V}_{usr}(\%)$	93.3338	93.8942	94.2563	94.5216	
	$\bar{V}_{ram}(\%)$	203.8262	201.5361	200.7799	200.3928	
Double MSPSort	T_{Q3}	2.1275	5.5136	11.6239	24.6005	$\beta=\text{ON}$ 24.6358
	$\bar{T}$	2.0899	5.4371	11.4603	24.2545	24.3857
	T_{Q2}	2.0806	5.4507	11.4952	24.2736	24.3834
	T_{Q1}	2.0452	5.3733	11.2931	23.8399	24.1348
	T_{IQR}	0.0823	0.1403	0.3308	0.7606	0.5010
	σ_T	0.0610	0.1154	0.2571	0.5938	0.4926
	$\bar{V}_{usr}(\%)$	93.3805	96.2759	97.2177	97.6680	97.7014
	$\bar{V}_{ram}(\%)$	100.8292	100.7939	100.4250	100.2632	
BQSort	T_{Q3}	2.0836	5.5412	11.7007	24.8755	
	$\bar{T}$	2.0442	5.4775	11.5933	24.6479	
	T_{Q2}	2.0424	5.4627	11.5666	24.6200	
	T_{Q1}	1.9991	5.4014	11.4318	24.3610	
	T_{IQR}	0.0845	0.1398	0.2689	0.5145	
	σ_T	0.0540	0.1134	0.2541	0.4409	
	$\bar{V}_{usr}(\%)$	97.9326	98.2525	98.4060	98.5310	
	$\bar{V}_{ram}(\%)$	101.9699	100.7939	100.3972	100.1997	
MWSort	T_{Q3}	1.9794	5.3865	11.6665	24.9317	
	$\bar{T}$	1.9667	5.3554	11.5684	24.8053	
	T_{Q2}	1.9639	5.3343	11.5659	24.6787	
	T_{Q1}	1.9504	5.2840	11.4709	24.4965	
	T_{IQR}	0.0290	0.1025	0.1956	0.4352	
	σ_T	0.0249	0.0826	0.1848	0.5598	
	$\bar{V}_{usr}(\%)$	90.9457	91.7825	92.3183	92.7510	
	$\bar{V}_{ram}(\%)$	201.9203	200.7772	200.3857	200.1931	

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

Optimizing MultiStack Parallel (MSP) Sorting Algorithm 23

Table 12 KPIs of of RAL-MSPSort vs BQSort vs MWSort for Uint32 and Double at various sizes N on R9-2920 system after 1000 trials

Alg.	KPI (Sec.)	200M	500M	1000M	2000M	2000M
Uint32 MSPSort	T_{Q3}	1.1615	3.0147	6.2208	12.8731	$\beta=ON$ 12.7956
	$\bar{T}$	1.1532	2.9871	6.1665	12.8096	12.6881
	T_{Q2}	1.1511	2.9720	6.1285	12.7004	12.6646
	T_{Q1}	1.1416	2.9378	6.0565	12.5842	12.5497
	T_{IQR}	0.0199	0.0765	0.1743	0.2889	0.2459
	σ_T	0.0194	0.0245	0.1192	0.2564	0.1976
	$\bar{V}_{usr}(\%)$	90.7344	93.3419	94.0156	94.0447	95.3851
	$\bar{V}_{ram}(\%)$	100.5634	100.3763	100.2624	100.2141	
BQSort	T_{Q3}	1.2849	3.3515	6.9910	14.8314	
	$\bar{T}$	1.2641	3.3027	6.8910	14.4880	
	T_{Q2}	1.2608	3.2872	6.8587	14.3754	
	T_{Q1}	1.2366	3.2211	6.7273	13.9707	
	T_{IQR}	0.0483	0.1304	0.2637	0.8607	
	σ_T	0.0398	0.1266	0.2883	0.7737	
	$\bar{V}_{usr}(\%)$	94.2977	94.8018	94.3827	91.0273	
	$\bar{V}_{ram}(\%)$	100.5719	100.2307	100.1184	100.0607	
MWSort	T_{Q3}	1.2690	3.1965	6.9421	14.6856	
	$\bar{T}$	1.2605	3.1748	6.7743	14.4544	
	T_{Q2}	1.2482	3.1424	6.8500	14.5239	
	T_{Q1}	1.2368	3.1253	6.4225	14.3433	
	T_{IQR}	0.0322	0.0712	0.5196	0.3423	
	σ_T	0.0351	0.0748	0.2940	0.3672	
	$\bar{V}_{usr}(\%)$	89.3874	91.3076	88.7198	85.9254	
	$\bar{V}_{ram}(\%)$	200.5459	200.2186	200.1084	200.0546	
Double MSPSort	T_{Q3}	1.7501	4.6861	9.7268	20.5472	$\beta=ON$ 20.2142
	$\bar{T}$	1.7335	4.6198	9.6218	20.3233	20.0028
	T_{Q2}	1.7302	4.6215	9.5645	20.2944	19.9662
	T_{Q1}	1.7137	4.5675	9.4322	20.0442	19.7528
	T_{IQR}	0.0364	0.1186	0.2946	0.5030	0.4614
	σ_T	0.0328	0.1771	0.3427	0.6693	0.4716
	$\bar{V}_{usr}(\%)$	87.3213	87.3098	86.9115	87.9475	87.9766
	$\bar{V}_{ram}(\%)$	100.3140	100.2474	100.2008	100.1848	
BQSort	T_{Q3}	1.8262	4.8709	10.2529	21.6878	
	$\bar{T}$	1.7977	4.7990	10.0812	21.3060	
	T_{Q2}	1.7821	4.7602	10.0035	21.0510	
	T_{Q1}	1.7562	4.6768	9.8058	20.6554	
	T_{IQR}	0.0700	0.1941	0.4471	1.0324	
	σ_T	0.0593	0.1729	0.4180	1.0467	
	$\bar{V}_{usr}(\%)$	92.0551	89.5123	86.6695	87.4219	
	$\bar{V}_{ram}(\%)$	100.2882	100.1184	100.0588	100.0297	
MWSort	T_{Q3}	1.5818	4.0025	8.9359	18.2426	
	$\bar{T}$	1.5700	4.0276	8.7317	18.0441	
	T_{Q2}	1.5659	3.9595	8.8775	18.0956	
	T_{Q1}	1.5542	3.9384	8.7910	17.9187	
	T_{IQR}	0.0276	0.0641	0.1449	0.3239	
	σ_T	0.0244	0.1598	0.3418	0.4057	
	$\bar{V}_{usr}(\%)$	86.3102	86.8115	82.9921	82.7472	
	$\bar{V}_{ram}(\%)$	200.2374	200.1084	200.0543	200.0269	

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า
ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

ประวัติผู้เขียน

ชื่อ-นามสกุล นายอภิสิทธิ์ รัตนารักษ์
วัน เดือน ปีเกิด 16 ธันวาคม 2531 ที่ฉะเชิงเทรา
ที่อยู่ 222/9-10 ต.หน้าเมือง อ.เมือง จ.ฉะเชิงเทรา 24000
โทร 0659635954

ประวัติการศึกษา 2554 วิศวกรรมศาสตรบัณฑิต สาขาวิชาวิศวกรรมคอมพิวเตอร์
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
2556 วิศวกรรมศาสตรมหาบัณฑิต สาขาวิชาวิศวกรรมคอมพิวเตอร์
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ความชำนาญเฉพาะด้าน 1.) การพัฒนาโปรแกรมแบบขนานโดยใช้โอเพนเอ็มพี
2.) การพัฒนาโปรแกรมด้วยหน่วยประมวลผลกราฟิกโดยใช้โอเพนซีแอล

ประสบการณ์การทำงานและผลงานวิจัย

พ.ศ. 2555-2556 ผู้ช่วยสอนวิชา Digital Circuit Laboratory วิทยาลัยนานาชาติ
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

พ.ศ. 2557-2561 อาจารย์ สาขาวิชาวิศวกรรมคอมพิวเตอร์ คณะเทคโนโลยีอุตสาหกรรม
มหาวิทยาลัยราชภัฏสวนสุนันทา

พ.ศ. 2561-ปัจจุบัน ผู้ช่วยศาสตราจารย์ สาขาวิชาวิศวกรรมคอมพิวเตอร์
คณะเทคโนโลยีอุตสาหกรรม มหาวิทยาลัยราชภัฏสวนสุนันทา

พ.ศ. 2561-2563 รองผู้อำนวยการฝ่ายวิจัยและพัฒนานวัตกรรมจัดการเรียนรู้
สำนักวิชาศึกษาทั่วไปและนวัตกรรมการเรียนรู้อิเล็กทรอนิกส์
มหาวิทยาลัยราชภัฏสวนสุนันทา