# อัลกอริทึมการแบ่งแบบขนานชนิดไพวอทคู่ สำหรับควิกซอร์ต

# PARALLEL DUAL PIVOT PARTITIONING ALGORITHMS FOR QUICKSORT


สุรพงศ์ เท่าเทียมตน
SURAPONG TAOTIAMTON

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
พ.ศ.2561
KMITL-2018-EN-M-070

# อัลกอริทึมการแบ่งแบบขนานชนิดไพวอทคู่
# สำหรับควิกซอร์ต

# PARALLEL DUAL PIVOT PARTITIONING ALGORITHMS FOR QUICKSORT


**สุรพงศ์ เท่าเทียมตน**

SURAPONG TAOTIAMTON


**วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต**

**สาขาวิชาวิศวกรรมคอมพิวเตอร์**

**คณะวิศวกรรมศาสตร์**

**สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง**

**พ.ศ.2561**


KMITL-2018-EN-M-070



เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

# PARALLEL DUAL PIVOT PARTITIONING ALGORITHMS FOR QUICKSORT

SURAPONG TAOTIAMTON

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENT FOR THE DEGREE OF

MASTER OF ENGINEERING IN COMPUTER ENGINEERING

FACULTY OF ENGINEERING

KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

2018

KMITL-2018-EN-M-070

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

COPYRIGHT 2018

FACULTY OF ENGINEERING

KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

**คณะวิศวกรรมศาสตร์**
**สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง**
**ใบรับรองวิทยานิพนธ์**

---

**หัวข้อวิทยานิพนธ์** อัลกอริทึมการแบ่งแบบขนานชนิดไพวอทคู่สำหรับควิกซอร์ต
**Thesis Title** Parallel Dual Pivot Partitioning Algorithms for QuickSort
**นักศึกษา** นายสุรพงศ์ เท่าเทียมตน
**รหัสประจำตัว** 58601093
**ปริญญา** วิศวกรรมศาสตรมหาบัณฑิต
**สาขาวิชา** วิศวกรรมคอมพิวเตอร์
**อาจารย์ที่ปรึกษาวิทยานิพนธ์** ผศ.ดร.สุรินทร์ กิตติธรกุล
**หมายเลขวิทยานิพนธ์** KMITL-2018-EN-M-070

| คณะกรรมการสอบวิทยานิพนธ์ | | ลายมือชื่อ |
|---|---|---|
| รศ.ดร.สุรพงศ์ | เอื้อวัฒนามงคล | |
| รศ.ดร.บุญธีร์ | เครือตราชู | |
| รศ.ดร.เกียรติกูล | เจียรนัยธนะกิจ | |
| ผศ.ดร.ชุติเมษฎ์ | ศรีนิลทา | |
| ผศ.ดร.สุรินทร์ | กิตติธรกุล | |

**วัน / เดือน/ ปี ที่สอบ** วันศุกร์ที่ 23 กุมภาพันธ์ พ.ศ. 2561 เวลา 10.30-12.30 น.
**สถานที่สอบ** ณ ห้องประชุม 3 ชั้น 5 อาคาร A

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

**คณะวิศวกรรมศาสตร์ รับรองแล้ว**

**(รองศาสตราจารย์ ดร. คมสัน มาลีสี)**
**คณบดี คณะวิศวกรรมศาสตร์**
**วันที่ 23 กุมภาพันธ์ พ.ศ. 2561**

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

| | |
|---|---|
| **หัวข้อวิทยานิพนธ์** | อัลกอริทึมการแบ่งแบบขนานชนิดไพวอทคู่สำหรับควิกซอร์ต |
| **นักศึกษา** | นายสุรพงศ์ เท่าเทียมตน |
| **รหัสประจำตัว** | 58601093 |
| **ปริญญา** | วิศวกรรมศาสตรมหาบัณฑิต |
| **สาขาวิชา** | วิศวกรรมคอมพิวเตอร์ |
| **พ.ศ.** | 2561 |
| **อาจารย์ที่ปรึกษาวิทยานิพนธ์** | ผศ.ดร.สุรินทร์ กิตติธรกุล |

## บทคัดย่อ

การจัดเรียงเป็นงานที่พบได้ในการประมวลผลข้อมูลและแอปพลิเคชันทั่วไปอีกจำนวนมาก ในปัจจุบันจำนวนข้อมูลมีขนาดใหญ่ขึ้นเรื่อยๆซึ่งทำให้การจัดเรียงข้อมูลนั้นใช้เวลานานมากขึ้น วิทยานิพนธ์ฉบับนี้จึงนำเสนอเสนออัลกอริทึมการแบ่งแบบขนานชนิดไพวอทคู่สำหรับควิกซอร์ตที่ทำงานบนคอมพิวเตอร์ชนิดมัลติคอร์/หลายคอร์ ชื่อ Hybrid Dual-Pivot Sort (HDPSort) and Parallel Dual-Pivot Sort (PDPSort) โดยอัลกอริทึมทั้งสองนี้สามารถทำการจัดเรียงข้อมูลได้ทุกชนิดและสามารถกำหนดฟังก์ชันในการเปรียบเทียบเองได้

อัลกอริทึม HDPSort และ PDPSort ถูกพัฒนาให้ทำงานแบบมัลติเทรดโดยใช้ไลบรารี OpenMP ทำให้ข้อมูลถูกแบ่งจนเหลือขนาดสั้นลง แล้วใช้แสตนดาร์ดเทมเพลทไลบรารีซอร์ตจัดเรียงแบบขนาน กล่าวคือ อัลกอริทึมการแบ่งแบบขนานทั้งสองนี้จะแบ่งข้อมูลให้มีขนาดสั้นลงได้เร็วขึ้นเพราะใช้ไพวอทจำนวน 2 ตัว จากการทดลองบนเครื่อง 4-คอร์ HyperThread Intel I7-2600, 8-core AMD FX-8320 และ 16-thread AMD R7-1700 บนระบบปฏิบัติการ Linux นั้นพบว่าสามารถเร็วได้กว่าแสตนดาร์ดเทมเพลทไลบรารีซอร์ตสูงสุด 2.99, 3.54 และ 6.13 เท่าตามลำดับ

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

| | |
|---|---|
| **Thesis** | Parallel Dual Pivot Partitioning Algorithms for QuickSort |
| **Student** | Mr.Surapong Taotiamton |
| **Student ID.** | 58601093 |
| **Degree** | Master of Engineering |
| **Program** | Computer Engineering |
| **Year** | 2018 |
| **Thesis Advisor** | Asst.Prof.Dr.Surin Kittitornkun |

# ABSTRACT

Sorting is one of the fundamental problems in data analytics and many other applications. At present, the amount of data is getting larger and larger. Thus, sorting data consumes much longer time. This thesis proposes two Parallel Dual-Pivot Partitioning Algorithms for C++ Standard Template Library (STL) Sort function on multi-core/many-core computers, namely Hybrid Dual-Pivot Sort (HDPSort) and Parallel Dual-Pivot Sort (PDPSort). In addition, they are compatible with STL Sort function thus supporting a wide variety of data types and user-define compare functions.

The HDPSort and PDPSort are developed and multithreaded by OpenMP library such that the resulting shorter subarrays can be STL Sorted in parallel. In other words, the smaller subarrays can be achieved sooner due to the proposed dual pivot partitioning algorithms. The experiments are conducted on a 4-core HyperThread Intel i7-2600, an 8-core AMD FX-8320 and a 16-thread AMD R7-1700 Ubuntu Linux systems. The achievable maximum Speedups by PDPSort are 2.99, 3.54 and 6.13 times faster than the STL Sort function, respectively.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

# กิตติกรรมประกาศ

วิทยานิพนธ์เล่มนี้สำเร็จได้ด้วยความกรุณาจากอาจารย์ที่ปรึกษา ผศ.ดร.สุรินทร์ กิตติธรกุล ที่ให้ความช่วยเหลือ ให้คำชี้แนะช่วยแก้ปัญหาตลอดจนให้ความรู้และประสบการณ์ที่ดีแก่ข้าพเจ้า

ขอขอบพระคุณบิดามารดาและพี่สาวที่ให้กำลังใจจนสามารถทำวิทยานิพนธ์เล่มนี้จนสำเร็จออกมาได้

สำหรับคุณงามความดีอันใดที่เกิดจากวิทยานิพนธ์ฉบับนี้ ข้าพเจ้าขอมอบให้กับบิดามารดาและครอบครัว ซึ่งเป็นที่รักและเคารพยิ่ง ตลอดจนครูอาจารย์ที่เคารพทุกท่านที่ได้ประสิทธิ์ประสาทวิชาความรู้และถ่ายทอดประสบการณ์ที่ดีให้แก่ข้าพเจ้า

สุรพงศ์ เท่าเทียมตน

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

# สารบัญ

**หน้า**




เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

# สารบัญ(ต่อ)

**หน้า**




เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

# สารบัญรูปภาพ

**รูปที่** **หน้า**



เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

# สารบัญรูปภาพ(ต่อ)

**รูปที่** **หน้า**



เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

# สารบัญตาราง

**ตารางที่** **หน้า**



เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

# บทที่ 1
# บทนำ

## 1.1 ที่มาและความสำคัญของปัญหา

ในปัจจุบันงานทางด้านคอมพิวเตอร์ต้องทำงานกับข้อมูลมีจำนวนมากขึ้นและขนาดใหญ่ขึ้น ตัวอย่าง เช่น การวิเคราะห์ข้อมูลขนาดใหญ่มาก (Big Data) การประมวลผลภาพ (Image processing) การทำเหมืองข้อมูล (Data mining) ซึ่งงานเหล่านี้จะต้องมีการจัดเรียงข้อมูล (Sorting) ซึ่งเป็นงานพื้นฐานของคอมพิวเตอร์ โดยการจัดเรียงข้อมูลขนาดใหญ่นั้นใช้เวลามาก ดังนั้นหากสามารถเพิ่มประสิทธิภาพการทำงานของจัดเรียงให้ทำงานได้เร็วขึ้นได้ก็จะเป็นการทำให้งานทางด้านคอมพิวเตอร์สามารถทำงานมีประสิทธิภาพสูงขึ้น

การเพิ่มประสิทธิภาพการจัดเรียงให้เร็วขึ้นนั้นสามารถทำได้โดยใช้ฮาร์ดแวร์แบบมัลติคอร์ เข้ามาช่วย แต่ก็ต้องทำการใช้การจัดเรียงแบบขนานจึงจะสามารถใช้งานมัลติคอร์ได้อย่างเต็มที่ ซึ่งการจัดเรียงแบบขนานนั้นมีความซับซ้อนและสามารถทำได้หลายวิธี เช่น การเพิ่มจำนวนไพวอทเพื่อเพิ่มจำนวนอาเรย์ที่ถูกแบ่งให้สามารถทำงานแบบขนานได้พร้อมกันมากขึ้น การทำการแบ่งข้อมูลเป็นช่วงๆในพื้นที่พิเศษแล้วทำการจัดเรียงแบบขนานพร้อมกัน เป็นต้น

ดังนั้นงานวิจัยนี้ได้เล็งเห็นว่าการจัดเรียงแบบขนานนั้นโดยการใช้วิธีการเพิ่มจำนวนไพวอทเพื่อเพิ่มจำนวนอาเรย์ที่ถูกแบ่งให้สามารถทำงานแบบขนานพร้อมกันได้นั้นเป็นวิธีที่เหมาะสมในการที่จะทำกาพัฒนาเนื่องจากใช้ทรัพยากรในการทำงานน้อยกว่าเมื่อเทียบกับวิธีอื่น

## 1.2 ปัญหา

1. การจัดเรียงข้อมูลที่มีขนาดใหญ่ใช้เวลาในการจัดเรียงเป็นเวลานาน
2. การจัดเรียงแบบขนานนั้นมีความซับซ้อนในการพัฒนา
3. การแบ่งข้อมูลด้วยไพวอทมากกว่าหนึ่งตัว

## 1.3 วัตถุประสงค์ของการศึกษา

1. เพื่อศึกษาการจัดเรียงด้วยไพวอทมากกว่าหนึ่งตัว
2. เพื่อเพิ่มประสิทธิภาพในการจัดเรียงแบบขนานบนเครื่องคอมพิวเตอร์ชนิดมัลติคอร์
3. เพื่อศึกษาปัจจัยที่มีผลกับประสิทธิภาพในการจัดเรียงแบบขนาน

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

## 1.4 โครงสร้างของเอกสารงานวิจัย

1. บทนำ
2. ทฤษฎีและงานวิจัยที่เกี่ยวข้อง
3. Parallel Hybrid Dual Pivot Sorting Algorithm
4. Parallel Dual-Pivot Quick Sort Algorithm
5. การทดลองและผลการทดลอง
6. สรุปและสิ่งที่จะพัฒนาต่อไป



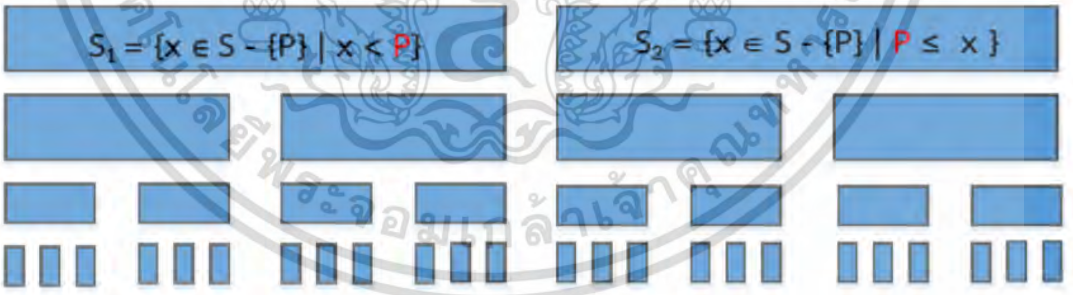
เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

# บทที่ 2
# ทฤษฎีและงานวิจัยที่เกี่ยวข้อง

ในปี ค.ศ. 1959 Tony Hoare ได้ทำการคิดค้นอัลกอริทึมควิกซอร์ต (QuickSort) โดยใช้หลักการ Divide & Conquer ชนิดไพวอทเดี่ยว และต่อมา Nico Lomuto ได้คิดค้นควิกซอร์ตที่ทำการแบ่งแบบ Lomuto ขึ้นมา แต่อัลกอริทึมที่คิดค้นขึ้นมานี้ยังใช้ไพวอทเดี่ยวในการแบ่งอยู่ จนถึงปี ค.ศ. 2009 Yaroslavskiy ได้ทำการคิดค้น Dual-Pivot Quicksort algorithm ซึ่งมีการใช้ไพวอทคู่ในการทำงานซึ่งทำให้ควิกซอร์ตทำงานได้เร็วขึ้น

## 2.1 Single Pivot QuickSort

การจัดเรียงข้อมูลนั้นมีหลายอัลกอริทึม (Algorithm) หนึ่งในนั้นคือการจัดเรียงแบบควิกซอร์ต (QuickSort) ซึ่งจะให้ประสิทธิภาพโดยเฉลี่ยเป็น $O(n \log n)$ ในการเรียงข้อมูลจำนวน n ตัว ส่วนในกรณีที่เลวร้ายที่สุด (Worst case) ของการจัดเรียงให้ประสิทธิภาพได้เป็น $O(n^2)$ ซึ่งกรณีนี้จะเกิดขึ้นได้ยาก เมื่อทำการเปรียบเทียบกันแล้วการจัดเรียงด้วยวิธีควิกซอร์ตในทางปฏิบัติมักจะเร็วกว่าอัลกอริทึมตัวอื่นที่มี Big O ในการจัดเรียงที่ให้ประสิทธิภาพเป็น $O(n \log n)$ ยกตัวอย่างเช่น Marge sort และ Heap sort ซึ่งสาเหตุที่ควิกซอร์ตเร็วกว่านั้นเพราะควิกซอร์ต เป็นการจัดเรียงแบบ In-place ทำให้ไม่ต้องใช้พื้นที่ในการจัดเรียงข้อมูลเพิ่มในกระบวนการทำงาน โดยการทำงานของควิกซอร์ต มีแนวคิดดังรูปที่ 2.1 และมีขั้นตอนดังต่อไปนี้




**รูปที่ 2.1** แสดงแนวคิดการทำงานของ Quick sort

1. ทำการเลือกค่าไพวอท P โดยเลือกจากข้อมูลในอาเรย์ที่ต้องการแบ่ง (Partitioning)
2. ทำการแบ่งอาเรย์ออกเป็น 2 กลุ่ม โดยใช้อัลกอริทึม คือ
    2.1 กลุ่มที่มีค่าน้อยกว่าไพวอท P
    2.2 กลุ่มที่มีค่ามากกว่าหรือเท่ากับไพวอท P
3. ทำการเรียกใช้ Quicksort แบบ Recursive กับอาเรย์ที่ต้องการแบ่งจนเหลือความยาวน้อยกว่าเท่ากับ 2 ตัว

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

อัลกอริทึมการทำแบ่งที่ได้รับความนิยมมี 2 วิธี คือ 1. Hoare และ 2. Lomuto โดย Hoare นั้นได้เปรียบในด้านจำนวนการเปรียบเทียบ (Comparison) และจำนวนการสลับข้อมูล (Swap) จึงใช้เวลาน้อยกว่าอัลกอริทึม Lomuto แต่อัลกอริทึม Lomuto สามารถใช้ cache ได้ดีกว่า


**รูปที่ 2.2** แสดงการแบ่งแบบ Hoare

การแบ่งแบบ Hoare นั้นมีขั้นตอนการแบ่งตามรูปที่ 2.2 คือ

1. ใช้ตัวแปร i สแกนจากด้านซ้ายไปทางด้านขวาโดยหาข้อมูลที่ตำแหน่ง i และมีค่ามากกว่าไพวอทเมื่อเจอให้ทำขั้นตอนต่อไป
2. ใช้ตัวแปร j สแกนจากด้านขวาไปด้านซ้ายหาข้อมูลที่ตำแหน่ง j ค่าที่มีค่าน้อยกว่าไพวอทเมื่อเจอทำขั้นตอนต่อไป
3. ทำการสลับค่าระหว่างที่ตัวแปร i และตัวแปร j
4. กลับไปทำข้อ 1 ใหม่อีกครั้งจนกว่าค่า j จะน้อยกว่าหรือเท่ากับ i
5. ตัวแปร j จะเป็นตัวแบ่ง


**รูปที่ 2.3** แสดงการแบ่งแบบ Lomuto

การแบ่งแบบ Lomuto นั้นมีขั้นตอนการแบ่งตามรูปที่ 2.3 คือ

1. เริ่มต้นตัวแปร i และ j ที่ด้านซ้ายสุดของอาเรย์
2. ใช้ตัวแปร j สแกนจากซ้ายไปขวาหาข้อมูลที่ตำแหน่ง j ที่มีค่าน้อยกว่าหรือเท่ากับไพวอท
3. เมื่อเจอทำการสลับค่าระหว่างตัวแปร i และ ตัวแปร j จากนั้นเพิ่มค่าตัวแปร i เป็น i+1
4. กลับไปทำข้อ 2 ใหม่อีกครั้งจนกว่าจะสุดอาเรย์ เมื่อสุดอาเรย์จะได้ตัวแปร i เป็นตัวแบ่ง

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

## 2.2 STL Sort

คือควิกซอร์ตที่บรรจุอยู่ใน Library : algorithm มาตรฐานของภาษา C++ โดยตัว STL Sort นั้นถูกออกแบบให้สามารถจัดเรียงข้อมูลได้หลายชนิด โดยผู้เขียนสามารถกำหนด Compare function ให้กับตัว STL sort ดังนั้นตัว STL Sort จึงสามารถทำงานยืดหยุ่น โดยตัว Function declaration ของ STL Sort คือ

*template <class RandomAccessIterator, class Compare>*
*void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);*

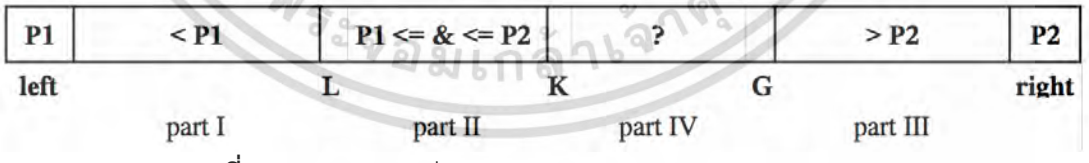
โดยความเร็วของ STLSort นั้นทำงานแบบอนุกรม (Sequential) มี BigO อยู่ที่ $O(n \log n)$ และเป็น Stable sort

## 2.3 Multiple Pivot Quicksort

Multiple pivot Quicksort คือควิกซอร์ตที่ใช้ไพวอท ตั้งแต่ 2 ตัวขึ้นไปในการทำงานเพื่อเพิ่มประสิทธิภาพในการให้เหนือกว่า Quick sort ที่ใช้ไพวอทเพียงตัวเดียว โดยตัวอย่าง Multiple Pivot Quicksort ที่มีการพัฒนาแล้วนั้นมีตัวอย่างดังต่อไปนี้

2.3.1 Dual-Pivot Quicksort algorithm (Vladimir Yaroslavskiy, 2009)

Dual-Pivot Quicksort algorithm [1] นั้นใช้ไพวอท 2 ตัวในการทำงานโดยมีแนวคิดการทำงานคล้ายๆกับควิกซอร์ตคือทำการแบ่งและ ทำการ Recursive อาเรย์ที่ทำการแบ่งแต่แตกต่างกันตรงส่วนการแบ่งโดยมีหลักการแบ่งดังรูปที่ 2.4 คือ




**รูปที่ 2.4** แสดงการแบ่งของ Dual-Pivot Quicksort algorithm

1. ทำการเลือกไพวอทขึ้นมา 2 ตัว คือ P1 และ P2 โดย P1 ต้องมีค่าน้อยกว่าหรือเท่ากับ P2

2. ใช้ตัวแปร 3 ตัวคือ L, K, G โดย L, K เริ่มต้นที่ด้านซ้ายสุดของอาเรย์และ G เริ่มต้นที่ด้านขวาสุดของอาเรย์

3. ใช้ตัวแปร K สแกนจากซ้ายไปขวาโดยหากเจอค่าน้อยกว่า P1 ให้ทำการสลับค่าที่ตัวแปร K และตัวแปร L จากนั้นเพิ่มค่าตัวแปร K ขึ้น 1 แต่ถ้าค่ามากกว่า P2 ให้สลับค่าระหว่างตัวแปร K และตัวแปร G จากนั้นลดค่า G ลง 1 หากไม่ตรงเงื่อนไขใดให้สแกนต่อไป

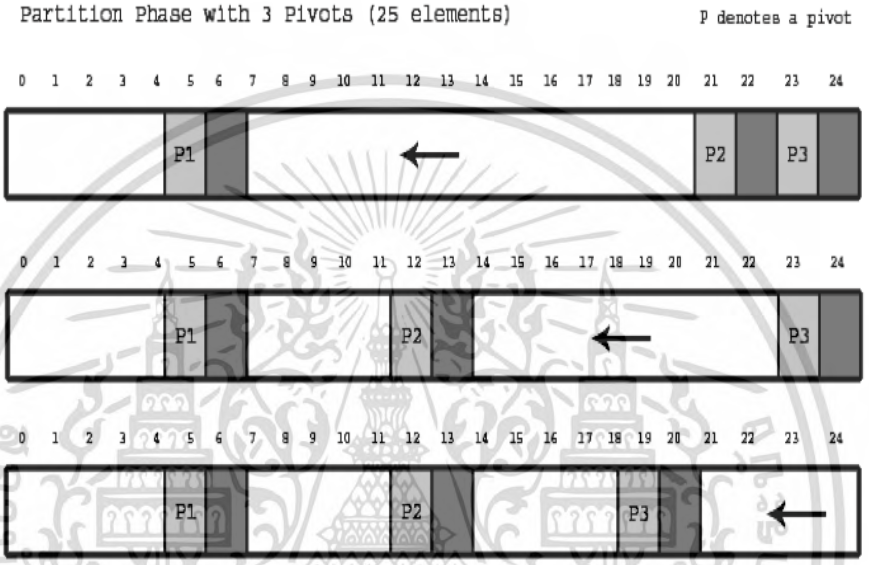
เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

4. ทำไปเรื่อยๆจนกว่าค่า K จะมากว่าหรือเท่ากับ G

5. ผลลัพธ์จะได้จากการแบ่งคืออาเรย์ 3 ส่วนดังรูปที่ 2.4 จากนั้นทำการเรียกใช้ Dual-Pivot Quicksort algorithm กับอาเรย์เหล่านั้น

2.3.2 Multiple Pivot Sort (Solehria & Jadoon, 2011)

Multi-Pivot Quicksort [2] นั้นใช้หลักการจัดเรียงเหมือนควิกซอร์ตคือทำการแบ่งจากนั้นทำการ Recursive เรียกใช้ฟังก์ชันกับอาเรย์ที่ทำการแบ่ง




**รูปที่ 2.5** แสดงการแบ่งของ Multiple Pivot Sort

ซึ่ง Multiple Pivot Sort นั้นสามารถใช้ไพวอทได้หลายตัวแล้วแต่จะกำหนดโดยมีขั้นตอนการแบ่งดังรูปที่ 2.5 คือ

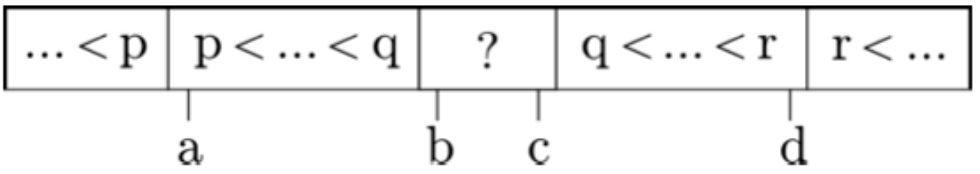
1. ทำการเลือกข้อมูลจำนวนหนึ่งจากอาเรย์เพื่อนำมาเป็นไพวอท จากนั้นนำมาจัดเรียงด้วย Insertion sort จากนั้นย้ายข้อมูลไปไว้ด้านหลังของอาเรย์

2. ทำการแบ่งโดยทำทีละไพวอท ตามรูปที่ 2.5 โดยจะเห็นว่าขนาดในการแบ่งจะน้อยลง

2.3.3 Multi-Pivot Quicksort (Kushagra et al, 2014)

Multi-Pivot Quicksort [3] นั้นใช้หลักการเดียวกับควิกซอร์ตคือทำการแบ่งและใช้ Recursive เรียกอาเรย์ที่ถูกแบ่งเหล่านั้นแต่จะมีขั้นตอนที่แตกต่างคือการทำแบ่งนั้นใช้ไพวอทจำนวน 3 ตัวในการทำการแบ่งโดยมีหลักการแบ่งดังรูปที่ 2.6 คือ

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้




**รูปที่ 2.6** แสดงการแบ่งของ Multi-Pivot Quicksort

1. ทำการเลือกไพวอทมาจำนวน 3 ตัวคือ p, q, r โดยค่า p ต้องน้อยกว่า q และ q ต้องน้อยกว่า r

2. ใช้ตัวแปร 4 ตัวคือ a, b, c, d โดย a, b นั้นเริ่มที่ด้านซ้ายสุดของอาเรย์และ c, d นั้นเริ่มที่ด้านขวาสุดของอาเรย์

3. ใช้ตัวแปร b สแกนจากซ้ายไปขวาโดยจากเจอค่าน้อยกว่า p ให้ทำการสลับตำแหน่งระหว่างค่า a และค่า b จากนั้นเพิ่มค่า a ขึ้นไป 1 และทำการสแกนต่อ แต่หากเจอค่าที่มากกว่า q ให้หยุดและทำขั้นตอนถัดไป

4.ใช้ตัวแปร c สแกนจากขวาไปซ้ายโดยหากเจอค่าที่มากกว่า r ให้ทำการสลับตำแหน่งระหว่างค่า c กับค่า d จากนั้นลดค่า d ลงไป 1 และทำการสแกนต่อ แต่หากเจอค่าที่มีค่าน้อยกว่า q ให้หยุดและทำขั้นตอนถัดไป

5. ทำการเปรียบเทียบค่าที่ตำแหน่งที่ b, c เทียบกับค่า p, q , r โดยมี 4 กรณีคือ

5.1 ค่าที่ตำแหน่ง b มากกว่าค่าที่ตำแหน่ง r และ ค่าที่ตำแหน่ง c น้อยกว่า p ทำการสลับตำแหน่งระหว่างค่าที่ตำแหน่ง b กับค่าที่ตำแหน่ง a จากนั้นสลับตำแหน่งระหว่างค่าที่ตำแหน่ง a กับค่าที่ตำแหน่ง c จากนั้นเพิ่มค่า a ขึ้น 1 จากนั้นทำการสลับค่าระหว่างค่าที่ตำแหน่ง c กับค่าที่ตำแหน่ง d จากนั้นเพิ่มค่า b ขึ้น 1 และลดค่า c กับ d ลง 1

5.2 ค่าที่ตำแหน่ง b มากกว่าค่าที่ตำแหน่ง r และ ค่าตำแหน่ง c มากกว่าหรือเท่ากับ p ทำการสลับตำแหน่งระหว่างค่าที่ตำแหน่ง b กับค่าที่ตำแหน่ง c จากนั้นทำการสลับค่าระหว่างค่าที่ตำแหน่ง c กับค่าที่ตำแหน่ง d จากนั้นเพิ่มค่า b ขึ้น 1 และลดค่า c กับ d ลง 1

5.3 ค่าที่ตำแหน่ง b น้อยกว่าหรือเท่ากับค่าที่ตำแหน่ง r และ ค่าที่ตำแหน่ง c น้อยกว่า p ทำการสลับตำแหน่งระหว่างค่าที่ตำแหน่ง b กับค่าที่ตำแหน่ง a จากนั้นสลับตำแหน่งระหว่างค่าที่ตำแหน่ง a กับค่าที่ตำแหน่ง c จากนั้นเพิ่มค่า a ขึ้น 1 จากนั้นเพิ่มค่า b ขึ้น 1 และลดค่า c ลง 1

5.4 ค่าที่ตำแหน่ง b น้อยกว่าหรือเท่ากับค่าที่ตำแหน่ง r ทำการสลับตำแหน่งระหว่างค่าที่ตำแหน่ง b กับค่าที่ตำแหน่ง c จากนั้นเพิ่มค่า b ขึ้น 1 และลดค่า c ลง 1

6. กลับไปทำข้อ 3 ใหม่จนกว่า b จะมากกว่า c ซึ่งจะแบ่งได้อาเรย์ 4 ชุด ตามรูปที่ 2.6

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

จากการศึกษา Single Pivot QuickSort และ Multiple Pivot QuickSort แล้วพบว่ามีการทำการเพิ่มไพวอทขึ้นมาในการทำการแบ่งเพื่อเพิ่มความเร็วในการจัดเรียงให้เร็วขึ้น แต่พื้นฐานการแบ่งยังตั้งอยู่บนพื้นฐานการแบ่งแบบ Hoare และ Lomuto ตัวอย่างเช่นว่าจะเป็นวิธีการแบ่งของ Dual-Pivot Quicksort algorithm (Vladimir Yaroslavskiy, 2009) ที่ใช้ตัวแปร K ในการไล่หาค่าและสลับค่ากับตัวแปร L, G ซึ่งคล้ายกับวิธีการแบ่งของ Lomuto เพียงแต่เพิ่มเงื่อนไขตำแหน่งการสลับให้มีมากขึ้น หรือ Multi-Pivot Quicksort (Kushagra et al, 2014) ที่ใช้ตัวแปร b,c ในการไล่หาค่าแล้วสลับค่าซึ่งคล้ายกับวิธีแบ่งของ Hoare แต่มีการเพิ่มเงื่อนไขและวิธีการสลับ

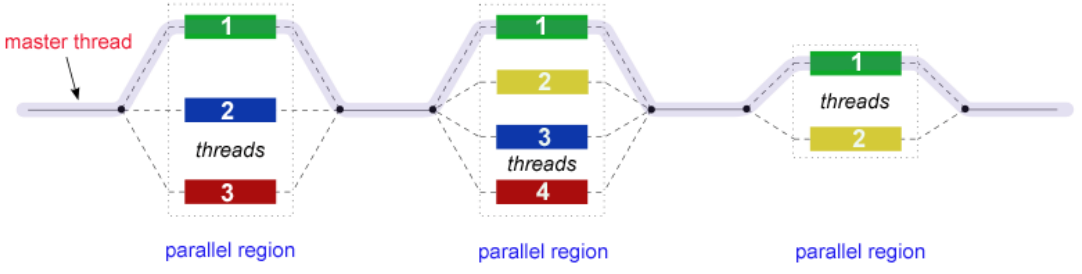
จาก Multiple Pivot QuickSort ที่ทำการศึกษานั้นพบว่ายังไม่มีการแบ่งแบบขนานอยู่เลย และวิธีการที่ศึกษานั้นยากที่จะดัดแปลงให้ทำการแบ่งแบบขนานได้เนื่องจากวิธีการนั้นไม่สามารถแบ่งงานให้สามารถทำงานแบบขนานได้พร้อมกัน ดังนั้นงานวิจัยนี้จึงสนใจที่จะทำการคิดค้นและออกแบบวิธีการแบ่งที่ใช้ไพวอทจำนวนมากกว่าหนึ่งตัวให้สามารถทำการแบ่งแบบขนานได้

## 2.4 OpenMP

OpenMP เป็นไลบรารี (Library) สำหรับการพัฒนาโปรแกรมที่ต้องการเพิ่มประสิทธิภาพให้สามารถทำงานแบบมัลติเทรด (Multithreading) บนเครื่องคอมพิวเตอร์ชนิด Shared Memory Multiprocessor หรือชนิดมัลติคอร์ (Multicore) การคำนวณแบบขนานนี้อาศัยการสร้าง Thread (Thread-base parallel programming) เพื่อให้เทรดทำงานได้อิสระมากขึ้นบนซีพียูคอร์ที่ยังว่างอยู่

OpenMP รองรับการพัฒนาโปรแกรมด้วยภาษา C, C++ และ Fortran โดยสามารถทำงานร่วมกับคอมไพเลอร์ต่างๆ เช่น GCC, Intel C/Fortran Compiler เป็นต้น เพื่อให้โปรแกรมสามารถทำงานบนระบบปฏิบัติการที่หลากหลาย เช่น Unix, Linux, Windows, Mac OS เป็นต้น

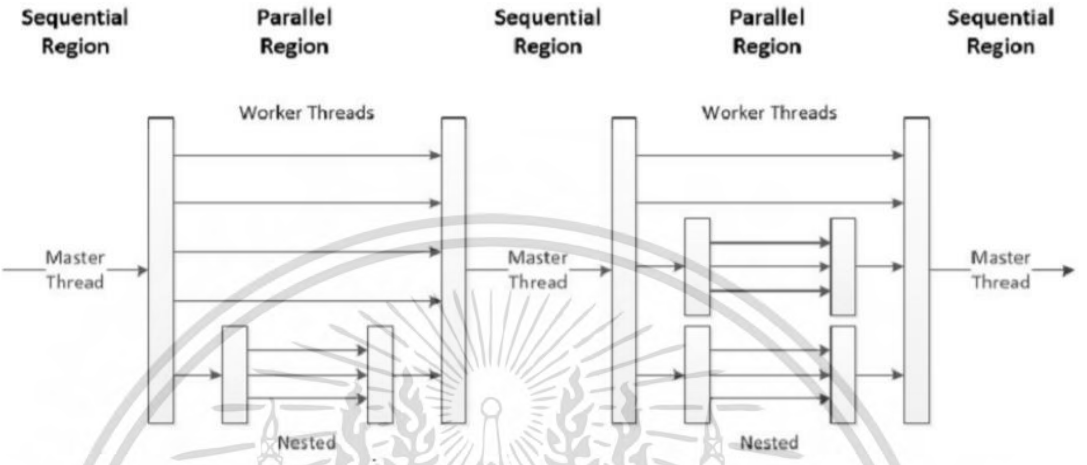
การทำงานของโปรแกรมที่ใช้ไลบรารี OpenMP มีรูปแบบ Fork-join model โดยในช่วง Runtime โปรแกรมจะเริ่มต้นจาก Master thread เพียง Thread เดียวก่อน หลังจากนั้น Master Thread จะสร้าง (Fork) Worker Thread จำนวนหนึ่ง ในช่วงที่ต้องการให้ทำงานแบบ Parallel ซึ่งเมื่องานในส่วนที่ต้องการเสร็จสมบูรณ์ Worker Thread ที่สร้างขึ้นมาเหล่านั้นจะถูก Master Thread ทำลาย (Join) แล้ว Master Thread จะทำงานส่วนที่เป็น Sequential ต่อไปดังรูปที่ 2.7




**รูปที่ 2.7** แสดงการทำงานของ Thread แบบ Fork-join

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

OpenMP นั้นสามารถสั่งให้โปรแกรมทำงานแบบขนานได้หลายรูปแบบตามลักษณะของงาน และ ตัวอย่างเช่น การสั่งให้ทำงาน Nested parallel กล่าวคือสามารถสั่งให้ทำงานแบบขนานภายใต้การทำงานที่เป็นขนานอยู่แล้วดังรูปที่ 2.8




**รูปที่ 2.8** แสดงการทำงานแบบ Nested parallel

งานวิจัยนี้ตัดสินใจเลือกใช้ OpenMP เป็นส่วนที่ทำให้โปรแกรมสามารถทำงานแบบขนานได้ นั้นเพราะ OpenMP นั้นเป็น API ที่เขียนง่ายและไม่ซับซ้อนเพียงทำการเพิ่มส่วนที่เป็นการกำหนดให้ทำงานแบบขนานก็สามารถทำงานแบบขนานได้แตกต่างจากว API อื่นที่ต้องทำการประกาศสร้าง Thread จัดการ Resource ของ Thread และทำลาย Thread เองทั้งหมด อีกทั้ง OpenMP เป็น API ที่ได้รับความนิยมจากกลุ่มผู้พัฒนาการเขียนโปรแกรมแบบขนานจึงทำให้สามารถหาข้อมูลและวิธีการพัฒนาด้วย OpenMP ได้ง่ายอีกด้วย และ OpenMP เป็น API ที่ได้รับการพัฒนามาอย่างต่อเนื่องและมีการวางแผนพัฒนาอย่างต่อเนื่อง

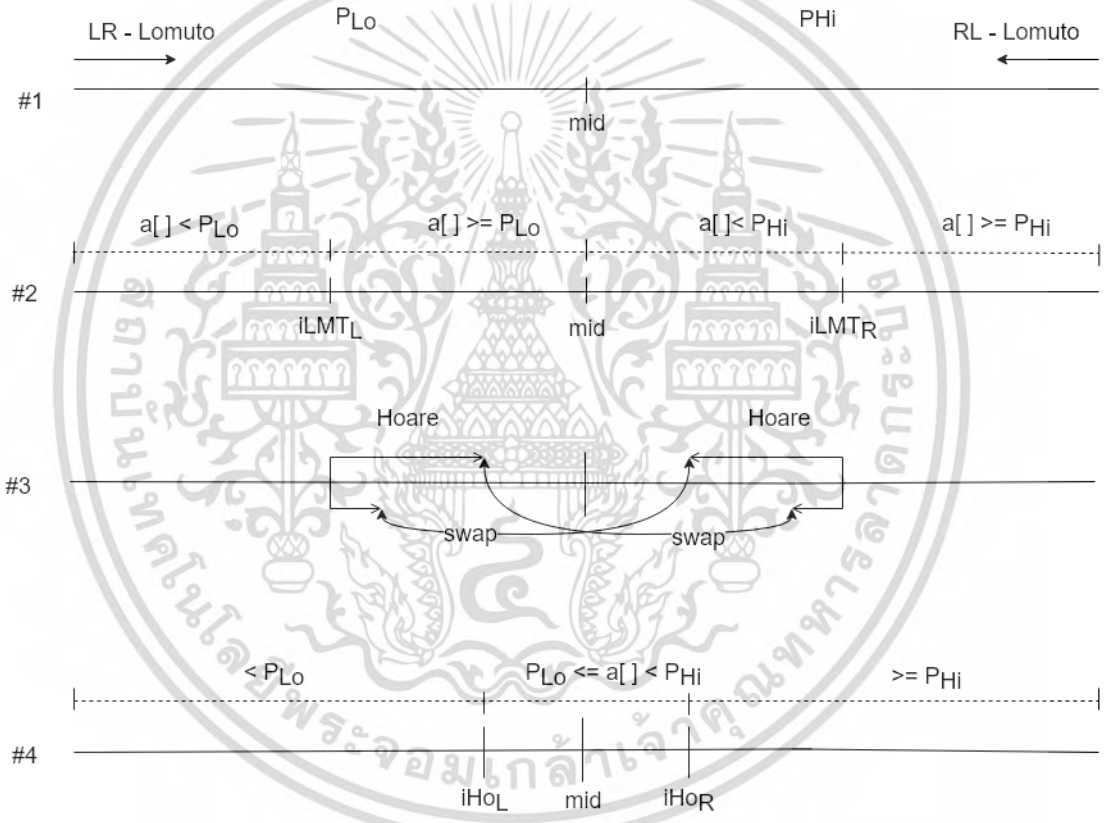
เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

# บทที่ 3

# Parallel Hybrid Dual Pivot Sorting Algorithm

## 3.1 หลักการทำงาน

Hybrid Dual Pivot Sorting (HDPSort) คือควิกซอร์ตที่ทำการใช้สองไพวอทในการแบ่ง (Partition) และใช้การแบ่งด้วยวิธี Hoare และวิธี Lomuto ร่วมกัน โดย HDP นั้นพัฒนาด้วยภาษา C++ และใช้ OpenMP เป็นไลบรารีในการพัฒนาให้สามารถทำงานได้แบบขนานได้ โดย HDPSort มีขั้นตอนการทำงานดังรูปที่ 3.1




**รูปที่ 3.1** แสดงแนวคิดและขั้นตอนการทำงานของ HDPSort

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

| Notation | Description |
|---|---|
| a[i] | Input data array of n elements |
| C | Number of CPU Cores |
| i, j | Left and right loop indices |
| $\mathrm{iHo_L}$ | Left Hoare index |
| $\mathrm{iHo_R}$ | Right Hoare index |
| $\mathrm{iLMT_L}$ | Left Lomuto Index |
| $\mathrm{iLMT_R}$ | Right Lomuto Index |
| $\mathrm{iP_{Lo}}$, $\mathrm{P_{Lo}}$ | The low pivot index and value |
| $\mathrm{iP_{Hi}}$, $\mathrm{P_{Hi}}$ | The hight pivot index and value |
| k | Number of Processor Sockets |
| K | $10^3$ elements |
| M | $10^6$ elements |
| N | Workload or Data Size (Elements) |
| S | Speed up |
| $\mathrm{T_{hdp}}$ | Run Time of Hybrid Dual Pivot Sort (Seconds) |
| $\mathrm{T_{stl}}$ | Run Time of STLSort (Seconds) |
| $\mathrm{U_{stl}}$ | STLSort Cutoff size (elements) |
| $\mathrm{U_{df}}$ | Depth-First Cutoff size (elements) |

**ตารางที่ 3.1** แสดงสัญลักษณ์ที่ใช้ในบทที่ 3

ขั้นตอนการทำงานของ HDPSort

1. ทำการแบ่งข้อมูลออกเป็น 2 ส่วนเท่าๆกัน โดยกำหนดให้ mid เป็นตำแหน่งกึ่งกลาง จากนั้นทำการหาค่าไพวอทในการแบ่งโดยทำการสุ่มหาดัชนีที่จะนำมาเป็นไพวอท โดยทำการหาค่า p ไพวอท 2 ค่า คือ $\mathrm{P_{Lo}}$ และ $\mathrm{P_{Hi}}$ โดย $\mathrm{P_{Hi}} > \mathrm{P_{Lo}}$ จากนั้นทำแบ่งแบบขนานพร้อมกันทั้งซ้ายและขวา โดยใช้ OpenMP Task ของ OpenMP ในการทำงาน โดยฝั่งซ้ายจะใช้ $\mathrm{P_{Lo}}$ เป็นไพวอท และใช้ LeftRight-Lomuto (LR-Lmt) ในการแบ่ง และ ฝั่งขวาจะใช้ $\mathrm{P_{Hi}}$ เป็นไพวอทและ RightLeft-Lomuto (RL-Lmt) ในการแบ่งโดยขั้นตอนนี้จะตรงกับรูปที่ 3.1 แถวที่ 1

2. หลังจากขั้นตอนที่ 1 จะแบ่งข้อมูลออกเป็นอาเรย์ 3 ส่วนตามรูปที่ 3.1 แถวที่ 2 คือ

1. $\mathrm{a[\,] < P_{Lo}}$ อยู่ทางด้านซ้าย
2. $\mathrm{P_{Lo} \le a[\,] < P_{Hi}}$ อยู่ตรงกลาง

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

3. $a[\,] \geq P_{Hi}$ อยู่ทางด้านขวา

3. เนื่องจากอาเรย์ที่อยู่ตรงกลางซึ่งมีอาจจะมีค่าที่น้อยกว่า $P_{Lo}$ และค่าที่มีค่ามากกว่า $P_{Hi}$ ปะปนอยู่ในอาเรย์ตรงกลางซึ่งเกิดจากการแบ่งโดยการแบ่งครึ่ง ดังนั้น อัลกอริทึมต้องทำการ แบ่งตรงกลางนั้นโดยใช้ Hoare partition โดยขอบเขตในการแบ่งคือ $iLMT_L$ ไปจนถึง $iLMT_R$ เป็นจำนวนสองครั้ง ดังนี้

A.ใช้ $P_{Lo}$ เป็นไพวอท ในการแบ่ง เมื่อทำการแบ่งเสร็จสิ้นจะได้ $iHo_L$ เป็นตำแหน่งแบ่งอาเรย์

B. ใช้ $P_{Hi}$ เป็นไพวอทในการแบ่ง เมื่อทำการแบ่งเสร็จสิ้นจะได้ $iHo_R$ เป็นตำแหน่งแบ่งอาเรย์

เมื่อทำการแบ่งตรงกลางเสร็จเรียบร้อยแล้วจะได้ผลลัพธ์ตามรูปที่ 3.1 แถวที่ 4

4. ทำการเรียกใช้ตัว HDPSort แบบ Recursive กับอาเรย์ที่ทำการแบ่งแล้ว โดยเรียกใช้แบบ Recursive โดยจะตัดสินใช้ Breadth first (BF) คือใช้ OpenMp Task สร้าง Thread ใหม่ขึ้นมาเพื่อทำงาน HDP Sort ที่ถูกเรียกแบบ Recursive หรือ Depth-first (DF) คือใช้ Thread เดิมในการทำงาน PDPSort ที่ถูกเรียกแบบ Recursive โดยการใช้ขนาดของอาเรย์เป็นตัวกำหนดวิธีการ Scheduling ซึ่งหากขนาดของอาเรย์มีค่ามากกว่า $U_{df}$ จะใช้ Scheduling แบบ Breadth first (BF) แต่ถ้าไม่จะทำการเรียกใช้ Depth-first (DF)

5. ทุกครั้งที่ทำการเรียกใช้งาน HDPSort ตัว HDPSort จะทำการตรวจสอบขนาดของ อาเรย์ว่ามีขนาดน้อยกว่า $U_{stl}$ หรือไม่หากมีขนาดน้อยกว่าจะทำการเรียกใช้ STLSort หากมากกว่าทำการใช้วิธีการตามข้อ 1 ถึง 4

โดย HDPSort นั้นมีการออกแบบให้มี $U_{stl}$ ขึ้นมานั้นเนื่องจากหลังจากได้ทำการศึกษาเกี่ยวกับการทำงานแบบขนานพบว่าจะมีประสิทธิภาพเมื่อทำงานกับข้อมูลที่มีขนาดใหญ่มากๆ แต่เมื่อทำงานกับข้อมูลที่มีขนาดไม่ใหญ่มากแล้วจะมีประสิทธิภาพน้อยกว่าเมื่อเทียบกับการทำงานแบบธรรมดา ดังนั้นจึงได้ออกแบบให้ HDPSort นั้นมีค่า $U_{stl}$ เพื่อให้ HDPSort เลือกที่จะใช้การทำงานแบบขนานหรือแบบธรรมดากับขนาดข้อมูลที่ต้องการทำการจัดเรียงที่เหมาะสม

ในส่วนของการออกแบบให้มีค่า $U_{df}$ ขึ้นมานั้นเนื่องจากการศึกษาการใช้งาน OpenMP และการทำงานแบบขนานแล้วพบว่า เมื่อทำการสร้าง Thread ขึ้นมาในระดับหนึ่งแล้วประสิทธิภาพในการทำงานเร็วขึ้น แต่เมื่อเพิ่มจำนวน Thread ขึ้นมามากขึ้นเรื่อยๆประสิทธิภาพการทำงานลดลง ดังนั้นจึงได้ออกแบบ HDPSort ให้มีค่า $U_{df}$ มาเป็นเงื่อนไขในการสร้าง Thread ใหม่ขึ้นมาในการทำงาน เพื่อไม่ให้เกิดการสร้าง Thread มากเกินจำเป็นซึ่งจะทำให้ประสิทธิภาพต่ำลง หรือ สร้าง Thread ขึ้นมาจำนวนน้อยเกินไปทำให้ใช้ทรัพยากร Thread ได้ไม่เต็มประสิทธิภาพ

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

**ALGORITHM 1:** Pseudocode of *HDPSort* Algorithm

```
1  Function Main()
2  |  HDPSort(a, 0, N − 1)
3  EndFunction
4  Function HDPSort(a, beg, end)
5  |  if end − beg + 1 < U_stl then
6  |  |  OpenMP Task
7  |  |  STLSort(a, beg, end)      // Call STLSort
8  |  |  return
9  |  end
10 |  mid = (beg + end)/2
11 |  iP_Lo, iP_Hi = Sample(a, beg, end)
12 |  swap(a, iP_Lo, mid − 1), swap(a, iP_Hi, mid + 1)
13 |  iP_Lo = mid − 1, P_Lo = a[iP_Lo]
14 |  iP_Hi = mid + 1, P_Hi = a[iP_Hi]
15 |  OpenMP Task
16 |  iLMT_L=LR_Lmt(a, beg, iP_Lo − 1, P_Lo)
17 |  OpenMP Task
18 |  iLMT_R=RL_Lmt(a, iP_Hi + 1, end, P_Hi)
19 |  OpenMP Taskwait
20 |  swap(a, iLMT_L, iP_Lo), swap(a, iLMT_R, iP_Hi)
21 |  iHo_L=Hoare(a, iLMT_L, iLMT_R, P_Lo)
22 |  iHo_R=Hoare(a, iLMT_L, iLMT_R, P_Hi)
23 |  if end − beg + 1 > U_df then
24 |  |  OpenMP Task
25 |  |  HDPSort(a, beg, iHo_L − 1)          // left
26 |  |  OpenMP Task
27 |  |  HDPSort(a, iHo_L, iHo_R − 1)        // middle
28 |  |  OpenMP Task
29 |  |  HDPSort(a, iHo_R, end)              // right
30 |  end
31 |  else
32 |  |  HDPSort(a, beg, iHo_L − 1)          // left
33 |  |  HDPSort(a, iHo_L, iHo_R − 1)        // middle
34 |  |  HDPSort(a, iHo_R, end)              // right
35 |  end
36 EndFunction
37 Function Hoare(a, bb, ee, p)
38 |  i = bb, j = ee
39 |  while i < j do
40 |  |  while a[i] < p do
41 |  |  |  i++
42 |  |  end
43 |  |  while a[j] ≥ p do
44 |  |  |  j − −
45 |  |  end
46 |  |  swap(a, i, j)
47 |  end
48 |  return j
49 EndFunction
50 Function LR_Lmt(a, bb, ee, p)
51 |  for i = j = bb; i ≤ ee; i++ do
52 |  |  if a[i] < p then
53 |  |  |  swap(a, i, j++)
54 |  |  end
55 |  end
56 |  return j
57 EndFunction
58 Function RL_Lmt(a, bb, ee, p)
59 |  for i = j = ee; i ≥ bb; i − − do
60 |  |  if a[i] ≥ p then
61 |  |  |  swap(a, i, j − −)
62 |  |  end
63 |  end
64 |  return j
65 EndFunction
```

**รูปที่ 3.2** แสดง Pseudo code ของ HDPSort

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

## 3.2 การทดลอง

| Parameters | Values |
|---|---|
| Data type | Unsigned Int 32 |
| Distribution | Random |
| Workload Size n (M) | 100, 200, 300, 400, 500 |
| Threads t | 4, 8, 16, 32 |
| $U_{stl}$(K) | 200, 400, 600, 800 |
| $U_{df}$ (M) | 1, 2, 4, 6, 8 |
| Optimization | O2 |

**ตารางที่ 3.2** แสดง Parameter ที่ทำการทดลอง

| System<br>Machine<br>Code Name | I7-2600<br>Bare Metal<br>Sandy Bride | FX-8320<br>Bare Metal<br>Piledriver |
|---|---|---|
| Clock (GHz) | 3.40 | 3.50 |
| Cores (c) | 4 | 8 |
| HyperThread $(t_{max})$ | Yes (8) | No (8) |
| RAM (GB) | 32 | 32 |
| Technology | DDR3-1333 | DDR3-1866 |
| L1 I-Cache | 4 x 32KB 8-way | 4 x 64KB 2-way |
| L1 D-Cache | 4 x 32KB 8-way | 8 x 16KB 4-way |
| L2 Cache | 4 x 256KB 8-way | 4 x 2MB 16-way |
| L3 Cache $(K_{IB})$ | 8MB 16-way | 8MB 64-way |

**ตารางที่ 3.3** แสดงข้อมูลของเครื่องที่ทำการทดลอง

3.2.1 วิธีการทดลอง

1. ทำการทดลองจัดเรียงด้วย HDPSort โดยทำการทดลองโดยใช้ Parameter ในการทดลองตามตารางที่ 3.2 โดยในแต่ละชุดการทดลองนั้นจะทำการทดลอง 5 ครั้งโดยใช้โปรแกรม Perf [4] ซึ่งเป็นโปรแกรมที่ใช้ในการเก็บข้อมูลของ Hardware/Software และใช้ค่าเฉลี่ยจากการทดลองเป็นผลการทดลอง

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

2. ทำการทดลองจัดเรียงด้วย STLSort โดยทำการทดลองบนชุดการทดลองขนาด 100M - 500M ซึ่งเป็นชุดเดียวกันกับที่ทดลองจัดเรียงด้วย HDPSort โดยทำการทดลองแต่ละชุดการทดลอง 5 ครั้งโดยใช้โปรแกรม Perf ซึ่งเป็นโปรแกรมที่ใช้ในการเก็บข้อมูลของ Hardware/Software และใช้ค่าเฉลี่ยจากการทดลองเป็นผลการทดลอง

3. นำผลการทดลองของ HDPSort และ STLSort มาทำการวิเคราะห์

3.2.2 ค่าที่ใช้ในการวัดประสิทธิภาพ

3.2.2.1 Run Time T: คือค่าเวลาเฉลี่ยจากการทดลอง 5 ครั้งในการทำการจัดเรียงด้วย STLSort ซึ่งก็คือ $\mathbf{T_{stl}}$ และค่าเวลาเฉลี่ยจากการทดลอง 5 ครังในการทำการจัดเรียง HDPSort ซึ่งก็คือ $\mathbf{T_{hdp}}$

3.2.2.2 Speedup: คือค่าวัดประสิทธิภาพระหว่างสองอัลกอริทึม โดยค่านี้เป็นการบอกว่า HDPsort ที่ทำงานแบบขนานนั้นทำงานได้เร็วกว่าเป็นกี่เท่าของ STLSort ที่เป็น Sequential โดยสามารถคำนวณได้จากค่า $\mathbf{T_{stl}}$ กับ $\mathbf{T_{hdp}}$ โดยค่า Speed up สามารถคิดได้จาก $\mathbf{S = \frac{T_{stl}}{T_{hdp}}}$

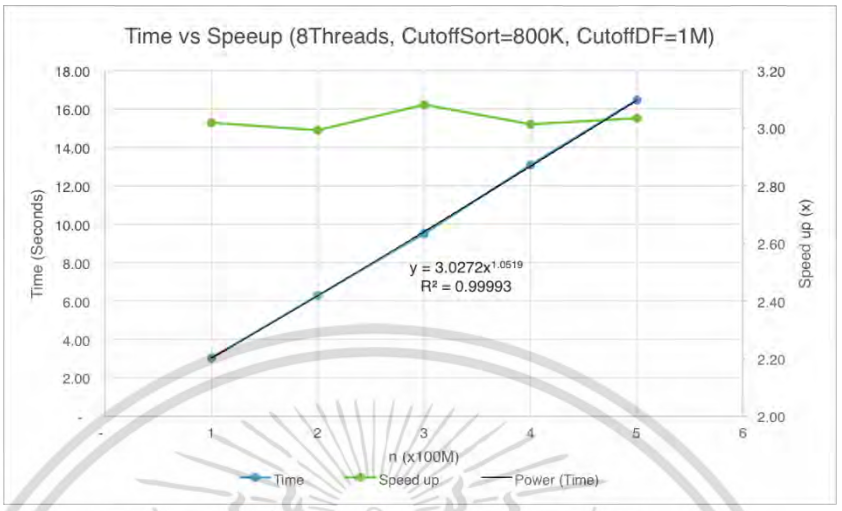
3.2.2.3 Instructions: คือ จำนวน Instruction ที่ถูกใช้การจัดเรียงข้อมูลด้วยจำนวน $\mathbf{t}$ thread บนชุดการทดลองขนาด $\boldsymbol{n}$ ตัว โดยจำนวน Instruction ที่ทำการวัดนั้นจะใช้โปรแกรมที่ชื่อว่า Perf ซึ่งเป็นโปรแกรมที่ใช้เก็บสถิติการทำงานของ software และ hardware

3.2.2.4 Branch Loads $\mathbf{B}$ and Branch Load Misses $\mathbf{B'}$ คือ ปริมาณการเกิด Branch Loads $\mathbf{B}$ และ Branch Load Misses $\mathbf{B'}$ ซึ่งเป็นตัวที่วัดประสิทธิภาพของอัลกอริทึมว่ามีประสิทธิภาพมากน้อยเพียงใด

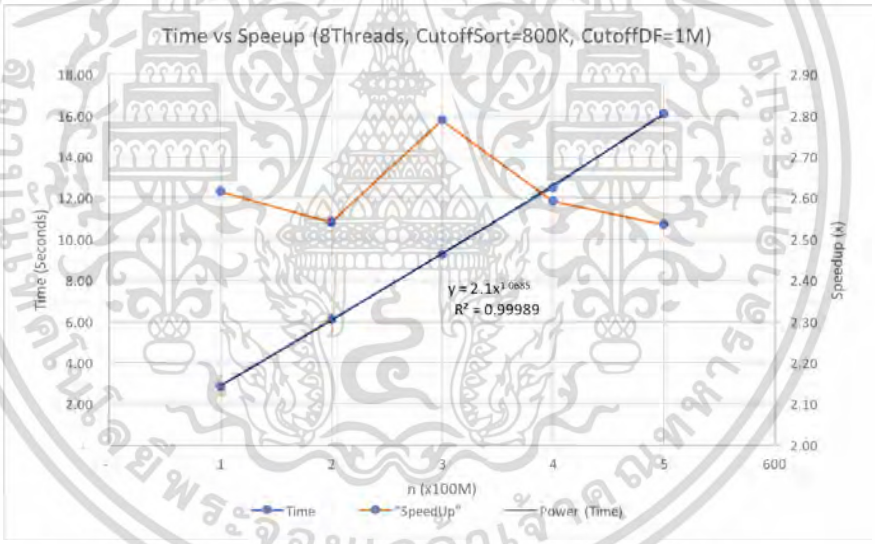
เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

## 3.3 ผลและการวิเคราะห์ผลการทดลอง

### 3.3.1 Best Run Time vs Speedup




**รูปที่ 3.3** กราฟแสดง Run Time vs Speedup โดยใช้ $\mathbf{U_{stl}}$ = 800K , $\mathbf{U_{df}}$ = 1M , $\mathbf{t}$ = 8 บนเครื่อง FX-8320



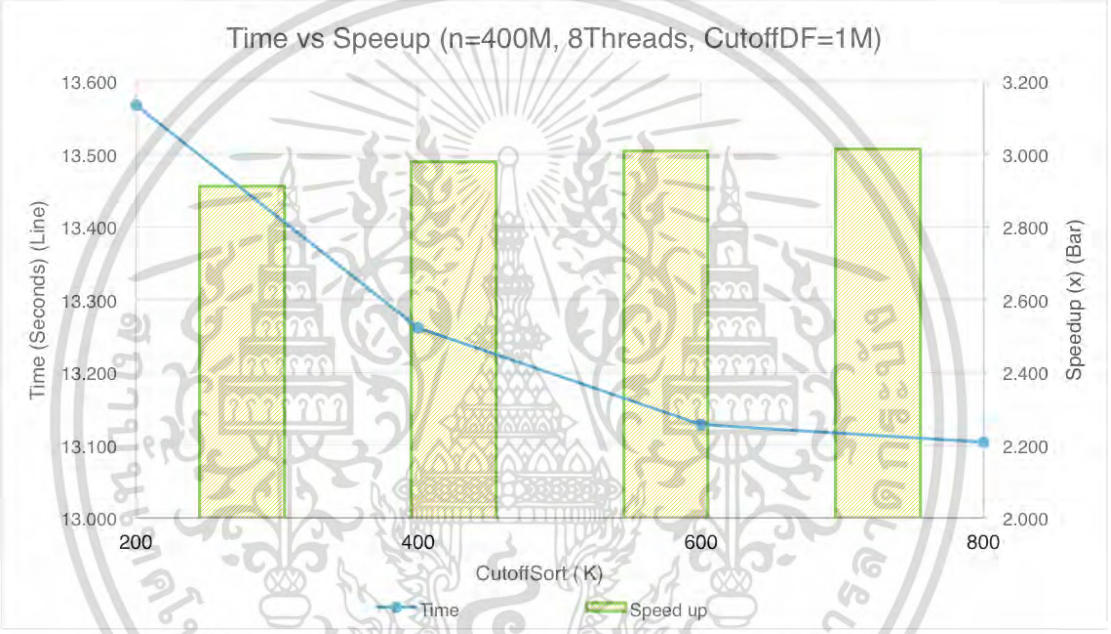

**รูปที่ 3.4** กราฟแสดง Run Time vs Speedup โดยใช้ $\mathbf{U_{stl}}$ = 800K , $\mathbf{U_{df}}$ = 1M , $\mathbf{t}$ = 8 บนเครื่อง I7-2600

จากรูปที่ 3.3 ซึ่งเป็นกราฟแสดงผลการทดลองเปรียบเทียบ Run Time ที่ดีที่สุดเทียบกับ Speed up สูงสุดบนเครื่อง FX-8320 โดยแกน X คือขนาดของจำนวนข้อมูลที่ต้องทำการจัดเรียงในขนาดตั้งแต่ 100 – 500 ล้านตัว และแกน Y เส้นสีน้ำเงินหมายถึงเวลาในการทำการจัดเรียงและเส้น

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

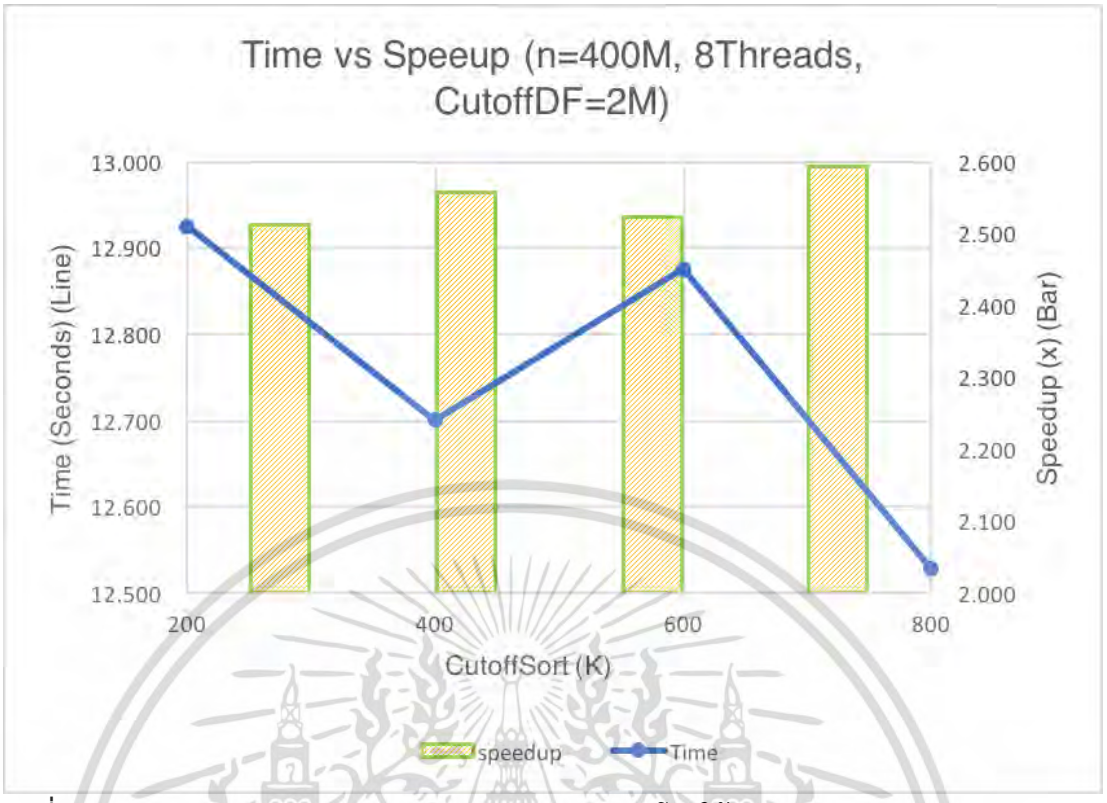
สีเขียวหมายถึง Speed up โดย Speed up ในการทำงานสูงสุดจะอยู่ที่ประมาณ 3.0 เท่าเมื่อเทียบกับ STL Sort และเมื่อนำผลการทดลองนี้มาสร้าง Linear Regression แล้วนั้นได้สมการเป็น $\mathrm{Y} = 3.0272\mathrm{x}^{1.0519}$ โดยมีค่า $\mathrm{R}^2 = 0.99993$ ซึ่งจากสมการ Liner Regression นั้นสอดคล้องกับค่า $n \log n$ ซึ่งเป็นค่า BigO ของ QuickSort เช่นเดียวกับ รูปที่ 3.4 ซึ่งเป็นกราฟแสดงผลการทดลองเปรียบเทียบ Run Time ที่ดีที่สุดเทียบกับ Speed up สูงสุดบนเครื่อง I7-2600 ที่นำผลการทดลองมาสร้าง Linear Regression จะได้สมการคือ $\mathrm{Y} = 2.1\mathrm{x}^{1.0685}$ โดยมีค่า $\mathrm{R}^2 = 0.99989$ ซึ่งค่าจากสมการนั้นสอดคล้องกับค่า $\mathrm{n} \log \mathrm{n}$ เช่นกัน ซึ่งทำให้สรุปได้ว่า BigO ของ HDPSort คือ $n \log n$

3.3.2 Speedup vs Sorting CutOff $\mathrm{U_{stl}}$




**รูปที่ 3.5** กราฟแสดง Speedup vs Sorting Cutoff $\mathrm{U_{stl}}$ โดยใช้ $\mathrm{U_{df}}$ = 1M, $\mathrm{t}$ = 8, n = 400M บนเครื่อง FX-8320

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

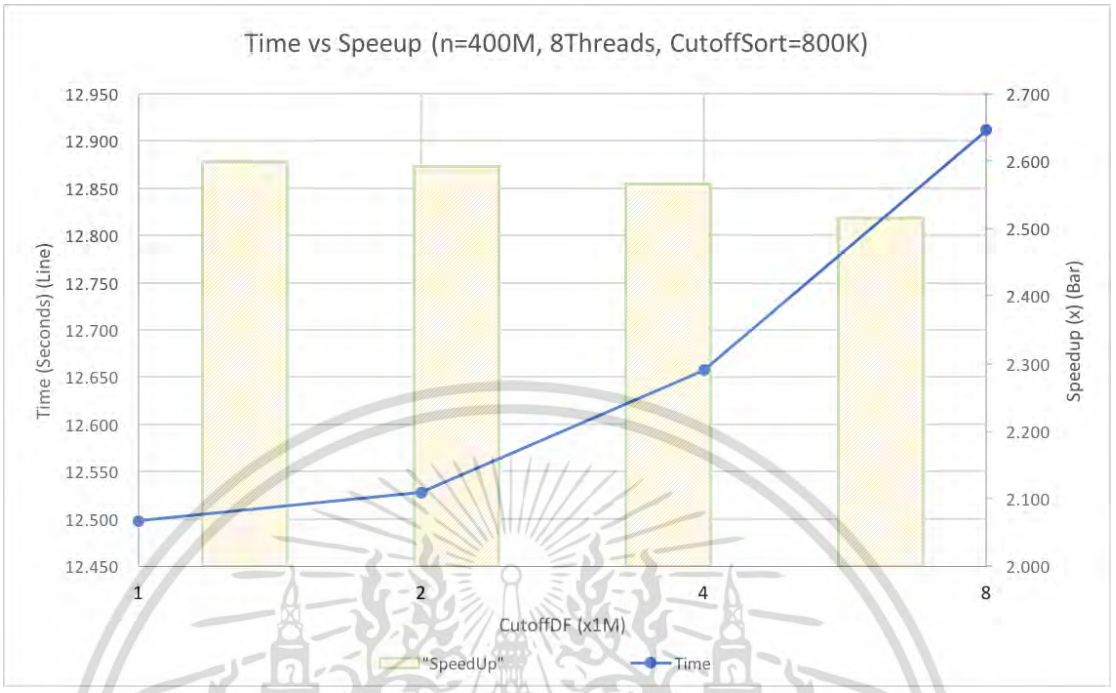



**รูปที่ 3.6** กราฟแสดง Speedup vs Sorting Cutoff $U_{stl}$ โดยใช้ $U_{df}$ = 2M, $t$ = 8, n = 400M บนเครื่อง I7-2600

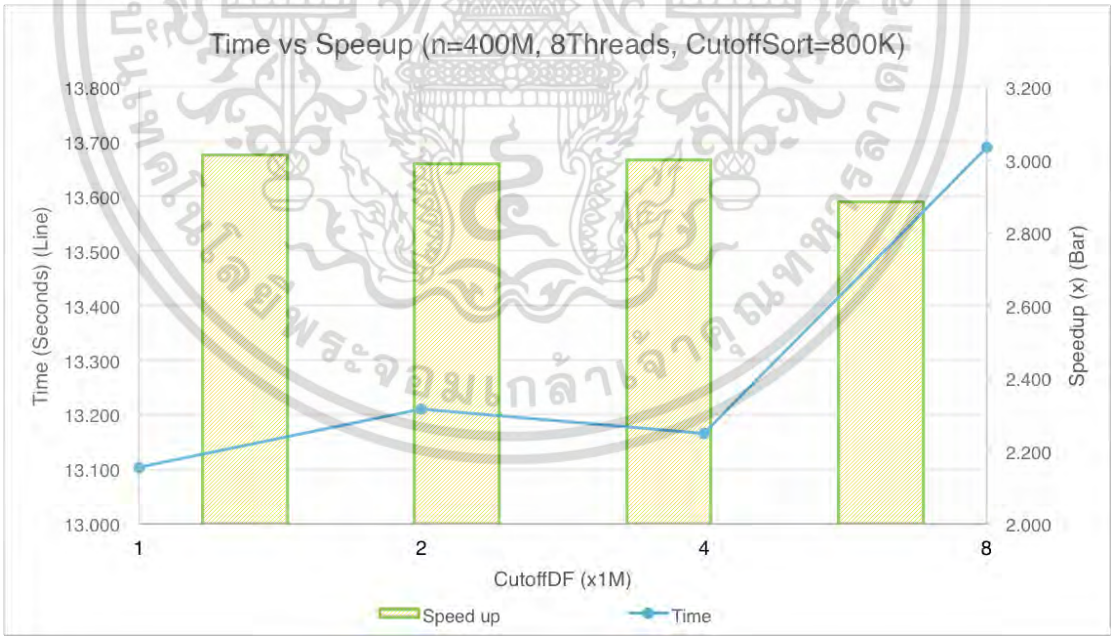
จากรูปที่ 3.5 เป็นกราฟเปรียบเทียบ Speed up กับ Sorting Cutoff $U_{stl}$ บนเครื่อง FX-8320 โดยใช้ $U_{stl}$ ในขนาด 200K – 800K เป็นค่าตัวแปรที่ทำการเปลี่ยนแปลงค่า ซึ่งจากผลการทดลองนั้นจะเห็นว่าเมื่อทำการเพิ่มขนาด $U_{stl}$ ขึ้นนั้นทำให้เวลาที่ใช้ในการจัดเรียงลดลงโดยดูได้จากเส้นสีน้ำเงินซึ่งทำให้ Speed up นั้นสูงขึ้นตามกราฟแท่ง เช่นเดียวกับรูปที่ 3.6 ที่เป็นกราฟเปรียบเทียบ Speed up กับ Sorting Cutoff บนเครื่อง I7-2600 ที่ Speed up มีแนวโน้มสูงขึ้นเมื่อทำการเพิ่มขนาด $U_{stl}$ ซึ่งทำให้เราสามารถสรุปได้ว่า HDPSort นั้นมีแนวโน้มที่จะทำงานได้เร็วขึ้นเมื่อเพิ่มขนาดของ $U_{stl}$ ซึ่งเกิดจากการจัดเรียงข้อมูลที่ขนาดชุดข้อมูลที่มีขนาดไม่ใหญ่มากนั้นการจัดเรียงแบบธรรมดานั้นทำงานได้เร็วกว่าขนานเนื่องจากไม่ต้องทำการสร้าง Thread และ Synchronization

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

3.3.3 Speedup vs Scheduling CutOff $\mathbf{U_{df}}$




**รูปที่ 3.7** กราฟแสดง Speedup VS Depth first Cutoff $\mathbf{U_{df}}$ โดยใช้ $\mathbf{U_{stl}}$ = 800 K, $\mathbf{t}$ = 8, n = 400M บนเครื่อง FX-8320




**รูปที่ 3.8** กราฟแสดง Speedup VS Depth first Cutoff $\mathbf{U_{df}}$ โดยใช้ $\mathbf{U_{stl}}$ = 800 K, $\mathbf{t}$ = 8, n = 400M บนเครื่อง I7-2600

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

รูปที่ 3.7 เป็นกราฟที่เปรียบเทียบ Speed up กับ Depth first Cutoff $U_{df}$ บนเครื่อง FX-8320 โดยใช้ $U_{df}$ ขนาด 1M – 4M เป็นตัวแปรที่ทำการเปลี่ยนแปลงค่า ซึ่งจากผลการทดลองนั้นแสดงให้เห็นว่าเมื่อทำการเพิ่มขนาดของ $U_{df}$ ให้มีขนาดที่ใหญ่ขึ้นเวลาที่ใช้ในการทำการจัดเรียงนั้นใช้เวลานานขึ้นและทำให้ Speedup ลดลง เช่นเดียวกับรูปที่ 3.8 ที่เป็นกราฟแสดง Speedup Vs Depth first Cutoff $U_{df}$ บนเครื่อง I7-2600 ที่เมื่อทำการเพิ่มขนาด $U_{df}$ ให้มีขนาดใหญ่ขึ้นแล้วทำให้เวลาที่ใช้ในการจัดเรียงเพิ่มขึ้นและ Speedup ลดลง โดยเกิดจากเมื่อขนาด $U_{df}$ มีขนาดใหญ่นั้นทำให้การเรียกใช้ Depth first schedule เกิดขึ้นเร็วทำให้อัลกอริทึมสร้าง Task เพื่อได้จำนวนน้อยกว่าจำนวน Task เมื่อการตั้งค่า $U_{df}$ ให้มีขนาดเล็กกว่า ซึ่งเมื่อสร้าง Task ได้น้อยทำให้เกิด Thread ว่างงานทำให้ใช้งานประสิทธิภาพได้ไม่เต็มที่

3.3.4 Speedup vs Other Ratios:

| n | S(x) | $I_{hdp}/ I_{stl}$ | $B_{hdp}/ B_{stl}$ | $B'_{hdp}/ B'_{stl}$ |
|---|---|---|---|---|
| 100M | 3.02 | 2.93 | 3.22 | 1.02 |
| 200M | 2.99 | 3.00 | 3.29 | 1.04 |
| 300M | 3.08 | 3.03 | 3.34 | 1.02 |
| 400M | 3.01 | 3.10 | 3.40 | 1.02 |
| 500M | 3.04 | 3.08 | 3.40 | 1.01 |

**ตารางที่ 3.4** แสดงการเปรียบเทียบค่าตัวแปรประสิทธิภาพที่ทำการทดลองโดยใช้ $U_{stl}$ = 800K, $U_{df}$ = 1M, $t$ = 8 บนเครื่อง FX-8320

| N | S(x) | $I_{hdp}/ I_{stl}$ | $B_{hdp}/ B_{stl}$ | $B'_{hdp}/ B'_{stl}$ |
|---|---|---|---|---|
| 100M | 2.62 | 3.10 | 3.22 | 1.02 |
| 200M | 2.54 | 3.12 | 3.29 | 1.04 |
| 300M | 2.79 | 3.14 | 3.34 | 1.02 |
| 400M | 2.59 | 3.21 | 3.40 | 1.02 |
| 500M | 2.54 | 3.23 | 3.40 | 1.01 |

**ตารางที่ 3.5** แสดงการเปรียบเทียบค่าตัวแปรประสิทธิภาพที่ทำการทดลองโดยใช้ $U_{stl}$ = 800K, $U_{df}$ = 1M, $t$ = 8 บนเครื่อง I7-2600

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

จากตารางที่ 3.4 และตารางที่ 3.5 ที่เป็นตารางแสดงข้อมูล Speedup, Instruction Ratio $\mathbf{I_{hdp}/\ I_{stl}}$ , Branch Loads Ratio $\mathbf{B_{hdp}/\ B_{stl}}$ และ Branch Loads misses Raito $\mathbf{B'_{hdp}/\ B'_{stl}}$ ที่ได้จากการทดลองทำการบนชุดข้อมูลขนาดตั้งแต่ 100M – 500 M บนเครื่อง FX-8320 และ I7-2600 ตามลำดับ โดยจากผลการทดลองนั้น Speed up นั้นมีแนวโน้มไปทางเดียวกับค่า Instruction Ratio และค่า Branch Loads Ratio นั่นคือเมื่อ Instruction Ratio หรือ Branch Loads Ratio มีค่าสูงขึ้นนั้นจะทำให้สามารถทำงานแบบ Parallel ได้สูงขึ้น อย่างไรก็ตามค่า Branch Loads misses Raito มีค่าอยู่ประมาณที่ 1.00 บนเครื่อง i7-2600 ตัว HDPSort นั้นมีความเร็วสูงกว่า STLSort ประมาณ 300% HDPSort นั้นมีค่า Branch Misprediction Ratio เพิ่มขึ้นอยู่ราวๆ 1-4 % ซึ่งเป็นค่าที่ต้องทำการ Tradeoff เมื่อใช้ HDPSort ซึ่งมันเป็นการ Tradeoff ที่ยอมรับได้เพื่อเพิ่มประสิทธิภาพให้สามารถทำงานแบบขนานได้

## 3.4 สรุปผลการทดลอง

HDPSort นั้นมีค่า Speed up อยู่ที่ประมาณ 3.00 เมื่อเทียบกับ STLSort โดยตัวแปรที่มีผลกับ HDPSort มีดังต่อไปนี้

1. $\mathbf{U_{stl}}$ เมื่อทำการเพิ่มขนาดของ $\mathbf{U_{stl}}$ ให้มีค่าสูงขึ้น HDPSort นั้นจะทำงานได้เร็วขึ้นดูได้จากรูปที่ 3.5 ที่เมื่อเพิ่มขนาดของ $\mathbf{U_{stl}}$ จะทำให้จัดเรียงได้เร็วขึ้นซึ่งเกิดจากการจัดเรียงชุดข้อมูลที่มีขนาดไม่ใหญ่มากนั้นการจัดเรียงแบบธรรมดานั้นทำงานได้เร็วกว่าแบบขนาน เนื่องจากไม่ต้องทำการสร้าง Thread และ Synchronization

2.$\mathbf{U_{df}}$ เมื่อทำการลดขนาดของ $\mathbf{U_{df}}$ ให้มีค่าลดลง HDPSort นั้นจะทำงานได้เร็วขึ้นดูได้จากรูปที่ 3.7 ที่เมื่อลดขนาดของ $\mathbf{U_{df}}$ จะทำให้การจัดเรียง ได้เร็วขึ้นโดยเกิดจากเมื่อขนาด $\mathbf{U_{df}}$ มีขนาดใหญ่นั้นจะทำให้ทำการใช้ Depth first schedule เกิดขึ้นเร็วทำให้ไม่สามารถสร้าง Task ให้ Thread ได้จำนวนมากเท่ากับการให้ $\mathbf{U_{df}}$ มีขนาดเล็กกว่า ซึ่งเมื่อสร้าง Task ได้น้อยทำให้เกิด Thread ว่างงานทำให้ใช้งานประสิทธิภาพได้ไม่เต็มที่

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

# บทที่ 4

# Parallel Dual-Pivot Quick Sort Algorithm

## 4.1 หลักการทำงาน

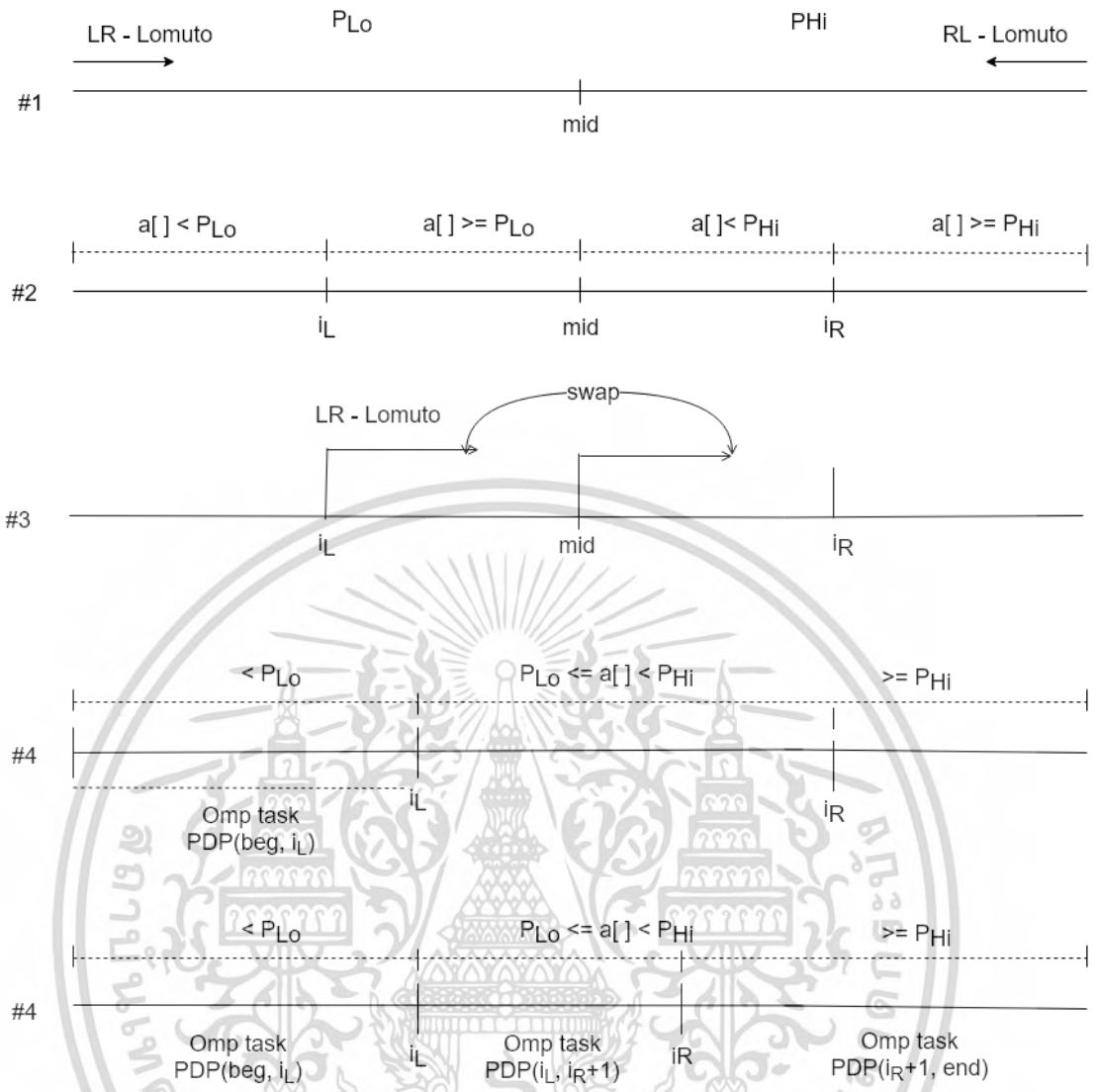
Parallel Dual-Pivot QuickSort (PDPSort) เป็นอัลกอริทึมที่พัฒนาโดยใช้หลักการแบ่งแบบ Lomuto ซึ่งการแบ่งแบบ Lomuto นั้นสามารถใช้งาน Cache ได้ดี โดยรูปที่ 4.1 แสดงแนวคิดและขั้นตอนการทำงานของ PDPSort และ ตารางที่ 4.1 แสดงสัญลักษณ์ที่ใช้บทนี้ โดยขั้นตอนการทำงานของ PDPSort นั้นใกล้เคียงกับ HDPSort แต่แตกต่างกันในการจัดการอาเรย์ที่อยู่ตรงกลางโดย PDPSort จะใช้วิธีการ Lomuto ส่วน HDPSort จะใช้วิธีการ Hoare โดยรายละเอียดจะอธิบายในขั้นตอนการทำงาน

PDPSort นั้นถูกพัฒนาด้วยภาษา C++ โดยสามารถดู Pseudo code ได้จากรูปที่ 4.2 ในส่วนของการเรียกใช้งาน PDPSort แบบ Recursive นั้นจะใช้ OpenMP Task เพื่อสร้าง Thread

ขั้นตอนการทำงานของ PDPSort

1. ทำการแบ่งข้อมูลออกเป็น 2 ส่วนเท่าๆกันโดยกำหนดให้ mid เป็นตำแหน่งกึ่งกลาง จากนั้นทำการหาไพวอทในการแบ่งโดยทำการสุ่มหาตำแหน่งที่จะนำมาเป็นไพวอท โดยทำการหาไพวอท 2 ตัว คือ $P_{Lo}$ และ $P_{Hi}$ โดย $P_{Hi} > P_{Lo}$ จากนั้นทำแบ่งแบบขนานพร้อมกันทั้งซ้ายและขวาโดยใช้ OpenMP Task ของ OpenMP ในการทำงาน โดยฝั่งซ้ายจะใช้ $P_{Lo}$ เป็นไพวอท และใช้ LeftRight-Lomuto (LR-Lmt) ในการแบ่ง และฝั่งขวาจะใช้ $P_{Hi}$ เป็นไพวอท และ RightLeft-Lomuto (RL-Lmt) ในการแบ่ง โดยขั้นตอนนี้จะตรงกับรูปที่ 4.1 แถวที่ 1

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้




**รูปที่ 4.1** แสดงแนวคิดและขั้นตอนการทำงานของ PDPSort

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

| Notation | Description |
|---|---|
| a[i] | Input data array of n elements |
| C | Number of CPU Cores |
| i, j | Left and right loop indices |
| $i_L$ | Left Lomuto index |
| $i_R$ | Right Lomuto index |
| $iP_{Lo}$, $P_{Lo}$ | The low pivot index and value |
| $iP_{Hi}$, $P_{Hi}$ | The hight pivot index and value |
| k | Number of Processor Sockets |
| K | $10^3$ elements |
| M | $10^6$ elements |
| N | Workload or Data Size (Elements) |
| S | Speed up |
| $T_{pdp}$ | Run Time of Dual Pivot QuickSort (Seconds) |
| $T_{stl}$ | Run Time of STLSort (Seconds) |
| $\tau$ | Number of Hardware Threads |
| $\tau_{max}$ | Maximum Hardware Threads |
| $U_{stl}$ | STLSort Cutoff size (elements) |
| $U_{df}$ | Depth-First Cutoff size (elements) |
| $V_{cpu}$ | CPU Utilization (non-Idle) <= 1.00 |

**ตารางที่ 4.1** แสดงสัญลักษณ์ที่ใช้ในบทที่ 4

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

2. หลังจากขั้นตอนที่ 1 จะแบ่งข้อมูลออกเป็นอาเรย์ 3 ส่วนตามรูปที่ 4.1 แถวที่ 2 คือ

1. $a[\,] < P_{Lo}$ อยู่ทางด้านซ้าย
2. $P_{Lo} \leq a[\,] < P_{Hi}$ อยู่ตรงกลาง
3. $a[\,] \geq P_{Hi}$ อยู่ทางด้านขวา

3.ทำการแบ่งที่อยู่ตรงกลางเพื่อย้ายค่าที่มีค่าน้อยกว่า $P_{Lo}$ ที่ยังปะปนอยู่ในอาเรย์ตรงกลาง โดยการใช้ $P_{Lo}$ เป็นไพวอท และใช้ LeftRight-Lomuto (LR-Lmt) ในการแบ่งโดยทำการเริ่มจุดสลับตำแหน่งที่ mid ตามรูปที่ 4.1 แถวที่ 3 จากนั้นเมื่อทำการแบ่งเสร็จแล้วตำแหน่งที่น้อยกว่า $i_L$ นั้น $a[i] < P_{Lo}$ ตามรูปที่ 4.1 แถวที่ 4

4. ทำการแบ่งอาเรย์ที่อยู่ตรงกลางเพื่อย้ายค่าที่มีค่ามากกว่า $P_{Hi}$ ที่ปะปนอยู่ในอาเรย์ตรงกลางโดยการใช้ $P_{Hi}$ เป็นไพวอท และ RightLeft-Lomuto (RL-Lmt) ในการแบ่งซึ่งเมื่อทำการ แบ่งเสร็จแล้วตำแหน่งที่มากกว่า $i_R$ นั้น $a[i] \geq P_{Hi}$ และตำแหน่งที่มีค่ามากกว่าเท่ากับ $i_L$ และน้อยกว่าเท่ากับ $i_R$ จะมีค่า $P_{Lo} \leq a[i] < P_{Hi}$ ตามรูปที่ 4.1 แถวที่ 5 และขณะเดียวกันอาเรย์ที่อยู่ด้านซ้ายนั้นสามารถทำการเรียกใช้ PDPSort เพื่อทำการจัดเรียงต่อได้เลยตามรูปที่ 4.1 แถวที่ 4 โดยจะอธิบายการเลือกว่า Breadth first(BF) และแบบ Depth-first (DF) ในส่วนถัดไป

5.ทำการเรียกใช้ PDPSort เพื่อทำการจัดเรียงอาเรย์ตรงกลางและอาเรย์ด้านขวา ตามรูปที่ 4.1 แถวที่ 5 โดยในขั้นตอนเรียกใช้ PDPSort แบบ Recursive ในขั้นตอนที่ 4 และ 5 นั้นเราจะตัดสินใช้ Breadth first (BF) คือใช้ OpenMP Task สร้าง Thread ใหม่ขึ้นมาเพื่อทำงาน PDPSort ที่ถูกเรียกแบบ Recursive หรือ Depth-first (DF) คือใช้ Thread เดิมในการทำงาน PDPSort ที่ถูกเรียกแบบ Recursive โดยการใช้ขนาดของอาเรย์ ซึ่งหากขนาดของอาเรย์มีค่ามากกว่า $U_{df}$ จะใช้ Breadth first (BF) แต่ถ้าไม่จะทำการเรียกใช้ Depth-first (DF)

6.ทุกครั้งที่ทำการเรียกใช้งาน PDPSort ตัว PDPSort จะทำการตรวจสอบขนาดของอาเรย์ว่ามีขนาดน้อยกว่า $U_{stl}$ หรือไม่หากมีขนาดน้อยกว่าจะทำการเรียกใช้ STLSort หากมากกว่าทำการใช้วิธีการตามข้อ 1 ถึง 5

การทำงานของ PDPSort มีขั้นตอนการทำงานคล้ายคลึงกับ HDPSort แต่แตกต่างกันตรงที่ HDPSort นั้นใช้วิธีการจัดการอาเรย์ตรงกลางโดยการใช้การแบ่ง แบบ Hoare แต่ PDPSort นั้นเลือกใช้ LefRight-Lomuto และ RightLeft-Lomuto เพราะหลังจากการทำ LeftRight-Lomuto แล้วนั้นสามารถใช้ OpenMP Task ทำการเรียกใช้ PDPSort ต่อได้เลยไม่ต้องการทำ RightLeft-Lomuto ซึ่งแตกต่างจาก HDPSort ที่ต้องทำการใช้ Hoare 2 ครั้งเพื่อจัดการแบ่งตรงกลางให้เรียบร้อยก่อนถึงจะเรียกใช้ HDPSort แบบ Recursive ได้ โดยที่แก้ไขแบบนี้ก็เพราะต้องการเพิ่มความเร็วในการจัดเรียงจากการทำงานแบบขนานที่ไม่ต้องรอให้ทำการแบ่งตรงกลางให้เสร็จทั้งก่อน

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

**ALGORITHM 1:** Pseudocode of *PDPSort* Algorithm

```
1  Function Main()
2  |  PDPSort(a, 0, N − 1)
3  EndFunction
4  Function PDPSort(a, beg, end)
5  |  if end − beg + 1 < U_stl then
6  |  |  OpenMP Task
7  |  |  STLSort(a, beg, end)      // Call STLSort
8  |  |  return
9  |  end
10 |  mid = (beg + end)/2
11 |  iP_Lo, iP_Hi = Sample(a, beg, end)
12 |  swap(a, iP_Lo, mid − 1),swap(a, iP_Hi, mid + 1)
13 |  iP_Lo = mid − 1, P_Lo = a[iP_Lo]
14 |  iP_Hi = mid + 1, P_Hi = a[iP_Hi]
15 |  OpenMP Task
16 |  i_L=LR_Lmt(a, beg, iP_Lo − 1, beg, P_Lo)
17 |  OpenMP Task
18 |  i_R=RL_Lmt(a, iP_Hi + 1, end, end, P_Hi)
19 |  OpenMP Taskwait
20 |  swap(a, i_L, iP_Lo),swap(a, i_R, iP_Hi)
21 |  i_L=LR_Lmt(a,mid,i_R,i_L,P_Lo)
22 |  if i_L − beg + 1 > U_df then
23 |  |  OpenMP Task
24 |  |  PDPSort(a, beg,i_L)              // left
25 |  end
26 |  else
27 |  |  PDPSort(a, beg,i_L)              // left
28 |  end
29 |  i_R=RL_Lmt(a,i_L,i_R,i_R,P_Hi)
30 |  if i_R − i_L > U_df then
31 |  |  OpenMP Task
32 |  |  PDPSort(a,i_L,i_R+1)             // middle
33 |  end
34 |  else
35 |  |  PDPSort(a,i_L,i_R+1)             // middle
36 |  end
37 |  if end − i_R + 1 > U_df then
38 |  |  OpenMP Task
39 |  |  PDPSort(a,i_R+1,end)             // right
40 |  end
41 |  else
42 |  |  PDPSort(a,i_R+1,end)             // right
43 |  end
44 EndFunction
45 Function LR_Lmt(a, bb, ee, j, p)
46 |  for i = bb; i ≤ ee; i++ do
47 |  |  if a[i] < p then
48 |  |  |  swap(a, i, j++)
49 |  |  end
50 |  end
51 |  return j
52 EndFunction
53 Function RL_Lmt(a, bb, ee, j, p)
54 |  for i = ee; i ≥ bb; i − − do
55 |  |  if a[i] ≥ p then
56 |  |  |  swap(a, i, j − −)
57 |  |  end
58 |  end
59 |  return j
60 EndFunction
```

**รูปที่ 4.2** แสดง Pseudo code ของ PDPSort

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

## 4.2 การทดลอง

| Parameters | Values |
|---|---|
| Data type | Unsigned Int 32 |
| Distribution | Random |
| Workload Size n (M) | 100, 200, 300, 400, 500 |
| Threads $\tau$ | 8, 16, 32 |
| $U_{stl}$(K) | 400, 600, 800 |
| $U_{df}$ (M) | 1, 2, 4 |
| Optimization | O2 |

**ตารางที่ 4.2** แสดง Parameterในการทดลอง

| System Name | I7-2600 Sandy Bridge | FX-8320 Piledriver | R7-1700 Ryzen |
|---|---|---|---|
| Clock (GHz) | 3.40 | 3.50 | 3.00 |
| Cores (c) | 4 | 8 | 8 |
| SMT $(t_{max})$ | Yes(8) | No(8) | Yes(16) |
| RAM (GB) | 32 | 32 | 32 |
| Technology | DDR3-1333 | DDR3-1866 | DDR4-2133 |
| L1 I-Cache | 4 x 32KB 8-way | 4 x 64KB 2-way | 8 x 64KB 4-way |
| L1 D-Cache | 4 x 32KB 8-way | 8 x 16KB 4-way | 8 x 32KB 8-way |
| L2 Cache | 4 x 256KB 8-way | 4 x 2MB 16-way | 8 x 512KB 8-way |
| L3 Cache | 8MB 64-way | 8MB 64-way | 2 x 8MB 16-way |

**ตารางที่ 4.3** แสดงสถาปัตยกรรม CPU และ RAM ที่ใช้ในการทดลอง

4.2.1 วิธีการทดลอง

1. ทำการทดลองจัดเรียงด้วย PDPSort โดยทำการทดลองโดยใช้ Parameter ในการทดลองตามตารางที่ 4.2 โดยในแต่ละชุดการทดลองนั้นจะทำการทดลอง 5 ครั้งโดยใช้โปรแกรม Perf ซึ่งเป็นโปรแกรมที่ใช้ในการเก็บข้อมูลของ Hardware/Software และใช้ค่าเฉลี่ยจากการทดลองเป็นผลการทดลอง

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

2. ทำการทดลองจัดเรียงด้วย STLSort โดยทำการทดลองบนชุดการทดลองขนาด 100M - 500M ซึ่งเป็นชุดเดียวกันกับที่ทดลองจัดเรียงด้วย HDPSort โดยทำการทดลองแต่ละชุดการทดลอง 5 ครั้งโดยใช้โปรแกรม Perf ซึ่งเป็นโปรแกรมที่ใช้ในการเก็บข้อมูลของ Hardware/Software และใช้ค่าเฉลี่ยจากการทดลองเป็นผลการทดลอง

3. นำผลการทดลองของ PDPSort และ STLSort มาทำการวิเคราะห์

4.2.2 ค่าในการวัดประสิทธิภาพ

4.2.2.1 Run time $\mathbf{T}$ : ค่านี้เป็นค่าเวลาที่ใช้ในการทำงานของแต่ละอัลกอริทึมโดยนำผลการทดลองทั้ง 5 ครั้งมาทำการหาค่าเฉลี่ยโดยเวลาที่ได้นั้นไม่รับรวมเวลาในการโหลดข้อมูลและ Overhead อื่นๆ โดย $\mathbf{T_{pdp}}$ คือเวลา Runtime ของ PDPSort และ $\mathbf{T_{stl}}$ เป็นเวลา Runtime ของ STLSort

4.2.2.2 Speedup$(x)$: เป็นค่าหลักที่ใช้ในการวัดประสิทธิภาพการทำงานโดยสามารถคำนวณได้จาก PDPSort ทำงานได้เร็วกว่าเป็นกี่เท่าของ STLSort โดยใช้การคำนวณจากค่า $\mathbf{T_{stl}}$ และ $\mathbf{T_{pdp}}$

$$S = \frac{T_{stl}}{T_{pdp}} \tag{4.1}$$

4.2.2.3 CPU Utilization$(\mathbf{V_{cpu}})$: ค่านี้ได้มาจาก /proc/stat ของ Linux ซึ่งทำการเก็บค่าสถิติการใช้งาน CPU โดยค่าที่ได้จะแบ่งออกเป็น user, system, idle โดยสามารถหาค่าได้จากสมการดังต่อไปนี้

$$V_{cpu} = \frac{1}{t_{max}} \sum_{i=0}^{t_{max}-1} V_{cpu_i} \tag{4.2}$$

เมื่อ $\mathbf{t_{max}}$ หมายถึงจำนวน Hardware thread และ $\mathbf{V_{cpu_i}}$ หมายถึงค่า non-idle CPU Utilization ของ Hardware thread ที่ *i* โดยค่า CPU Utilization ที่สูงหมายถึงการใช้เวลาในการใช้ CPU ในรูปแบบขนานได้น้อย

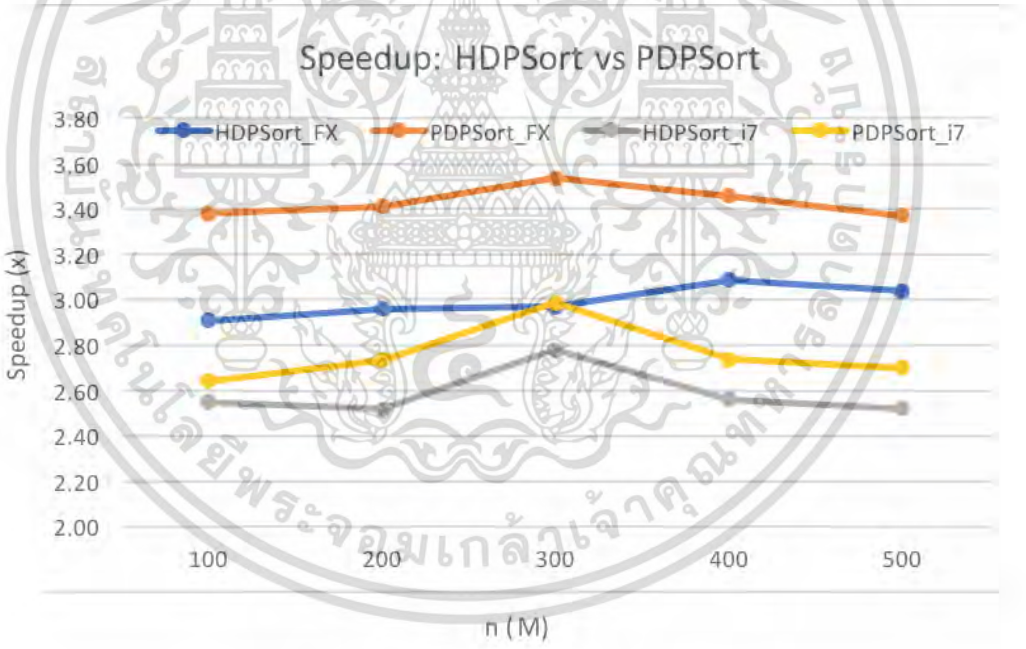
เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

## 4.3 ผลและวิเคราะห์ผลการทดลอง

ผลการทดลองเกี่ยวกับประสิทธิภาพของ PDPSort จะแสดงในมุมมองต่างๆ ดังต่อไปนี้

### 4.3.1 PDPSort vs HDPSort

ในการเปรียบเทียบ Speedup ของ PDPSort กับ HDPSort ซึ่งใช้ชุดข้อมูลแบบเดียวกัน โดยทดลองตั้งแต่ขนาด 100M – 500M และบน FX8320 and I7-2600 โดยผลการทดลองนั้นจะเห็นว่า PDPSort บน FX8320 ทำงานได้เร็วกว่า STLSort 3.3 ถึง 3.5 เท่า ซึ่งเร็วกว่า HDPSort ซึ่งทำงานได้เร็วกว่า STLSort 2.9 ถึง 3.1 เท่า และบน I7-2600 PDPSort ก็ทำงานได้ดีกว่า HDPSort ซึ่งดูได้จากรูปที่ 4.3 โดยเหตุผลที่ PDPSort ทำงานได้เร็วกว่าก็เพราะในขั้นตอนการจัดการแบ่งอาเรย์ตรงกลางของ PDPSort นั้นใช้ Lomuto partition โดยทำการเริ่มสลับตรงจุดกึ่งกลางทำให้ลดจำนวนการเปรียบเทียบลงได้จำนวนหนึ่ง อีกทั้งอาเรย์ทางด้านซ้ายที่ทำการจัดเสร็จแล้วสามารถใช้ OpenMP Task ทำงานไปพร้อมกับการจัดการแบ่งอาเรย์ที่เหลือได้

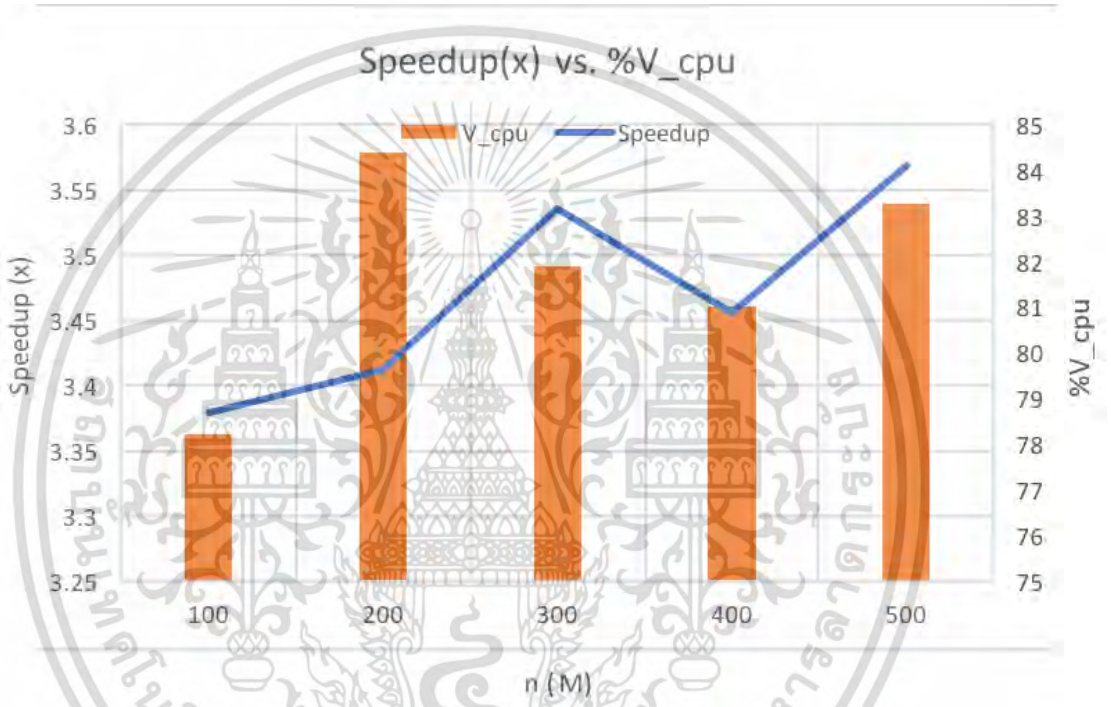



**รูปที่ 4.3** กราฟแสดง PDPSort vs HDPSort โดยใช้ N=100M-500M, $U_{stl}$= 800K, $\tau$= 8, $U_{df}$ = 1 M บนเครื่อง FX-8320 และ i7-2600

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

4.3.2 Speedup $\mathbf{S}$ vs CPU Utilization $\mathbf{V_{cpu}}$

ในการเปรียบเทียบค่า Speedup $\mathbf{S}$ กับ CPU Utilization $\mathbf{V_{cpu}}$ จากผลการทดลองที่ทดลองบน FX8320 ทดลองด้วยชุดข้อมูลขนาด 100M – 500M , $\tau = 16$ , $U_{df} = 1M$ , $U_{stl} = 800K$ โดยจากผลการทดลองพบว่าค่าจำนวนข้อมูลมีค่ามากขึ้นค่า Speedup ก็มีค่าสูงขึ้น อีกทั้งค่า CPU Utilization ก็เพิ่มขึ้นจาก 78% ไปเป็น 84% ซึ่งทำให้สรุปได้ว่าเมื่อค่า CPU Utilization ค่า Speedup ก็จะสูงขึ้นด้วยดังรูปที่ 4.4

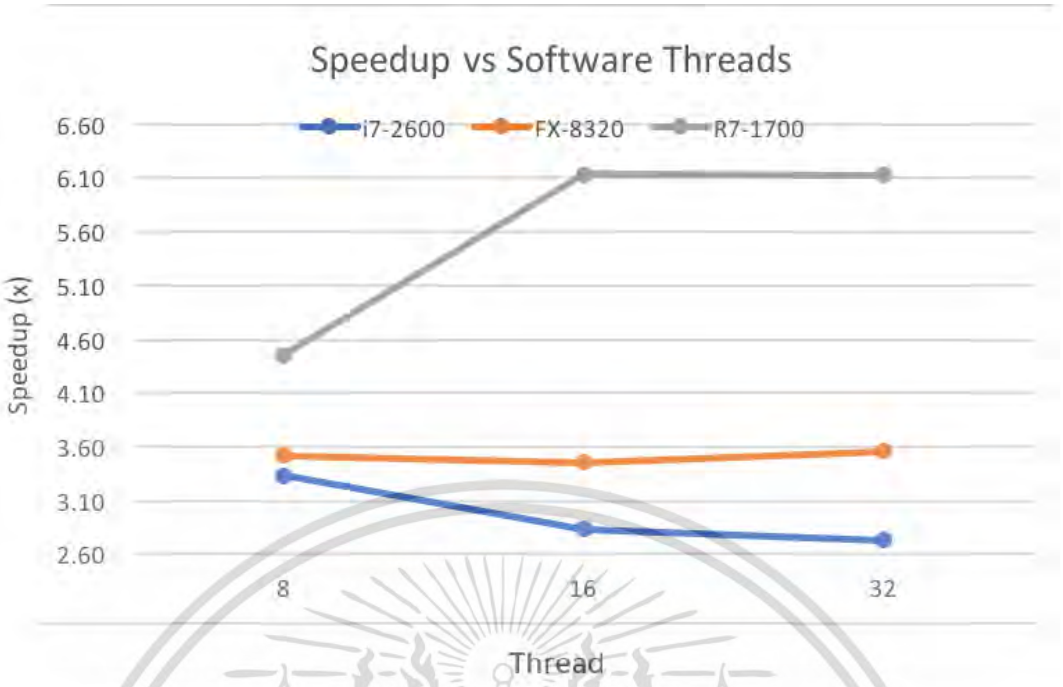



**รูปที่ 4.4** กราฟแสดง Speedup vs $\%V_{cpu}$ โดยใช้ N=100M-500M, $\tau = 16$, $U_{df} = 1M$ , $U_{stl} = 800K$ บนเครื่อง FX-8320

4.3.3. Speedup $\mathbf{S}$ vs Software Threads $\boldsymbol{\tau}$

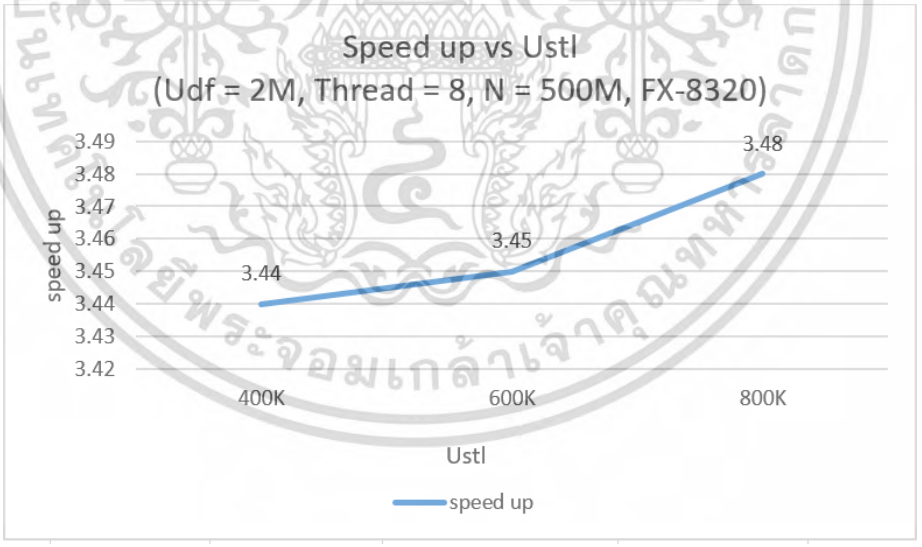
รูปที่ 4.5 แสดงให้เห็นแนวโน้มของ Speedup เมื่อเทียบกับ Software thread ในแต่ละเครื่อง โดย R7-1700 มีค่า Speed up สูงสุด ที่ $\boldsymbol{\tau}$ มีค่า 16 และ 32 ซึ่งมากกว่า $\mathbf{t_{max}}$ ซึ่งมี 8 Cores 16 Thread แต่ในทางตรงกันข้าม PDPSort ไม่ได้ทำการจัดการในกรณีที่ $\tau > t_{max}$ ซึ่งดูได้จากผลการทดลองบนเครื่อง I7-2600 ได้ดีทำให้ประสิทธิภาพไม่เท่ากับ R7-1700 โดยเมื่อทำการดูค่าที่วัดได้จาก Perf แล้วพบว่าการเพิ่ม Software thread ก่อให้เกิด Cache miss และ Data TLB store misses เพิ่มขึ้นซึ่งเกิดจากตัว Software thread ทำการใช้งาน CPU ร่วมกัน ทำให้เกิด Page Table access และ Page faults

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้



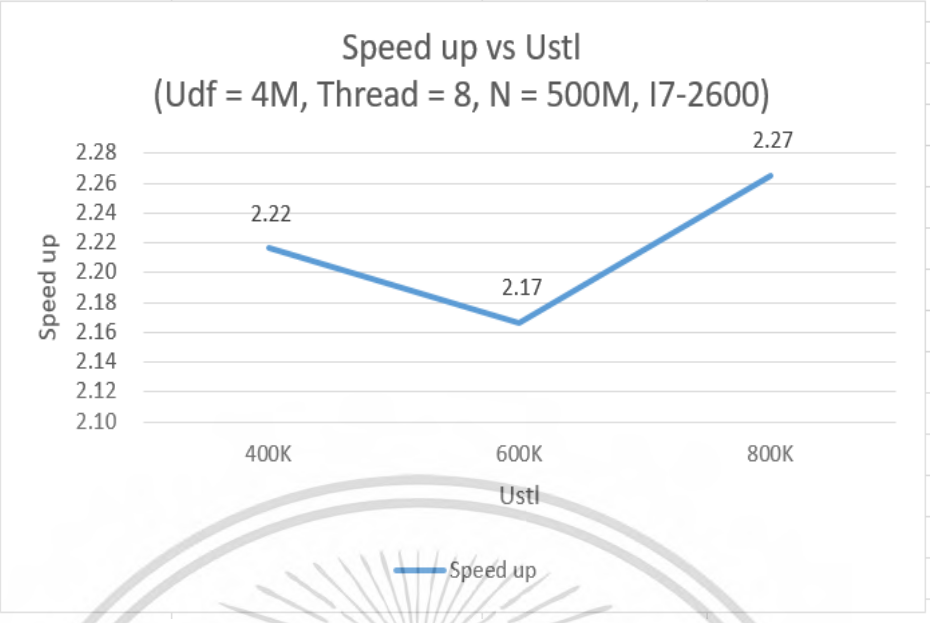

**รูปที่ 4.5** กราฟแสดง Speedup vs software Thread $\tau = 8, 16, 32$ , N=400M , $U_{stl}$ = 800K, $U_{df}$ = 1M

4.3.4. Speedup $S$ vs $U_{stl}$




**รูปที่ 4.6** กราฟแสดง Speed up vs $U_{stl}$ โดยใช้ N = 500M, $U_{df}$ = 2M, $\tau$ = 8 บนเครื่อง Fx-8320

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

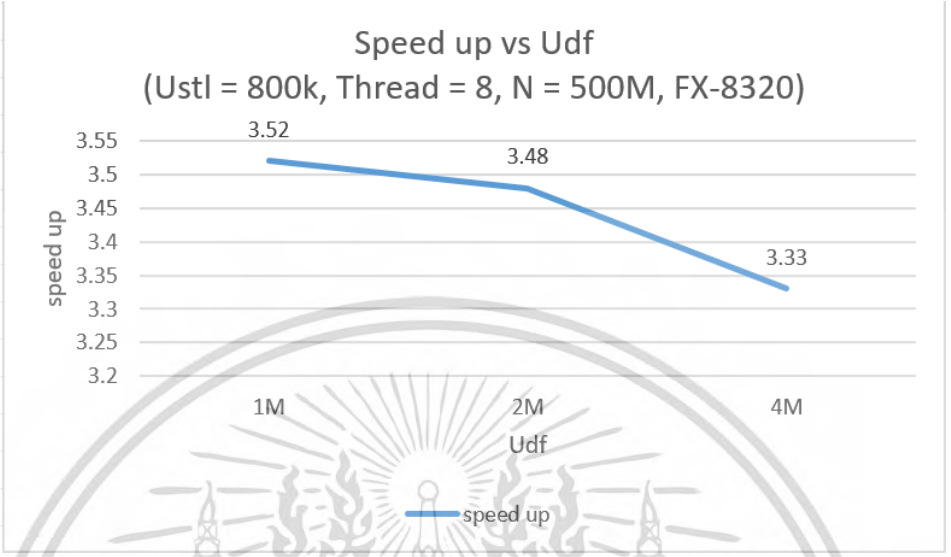



**รูปที่ 4.7** กราฟแสดง Speed up vs $U_{stl}$ โดยใช้ N = 500M, $U_{df}$ = 4M, $\tau$ = 8 บนเครื่อง I7-2600

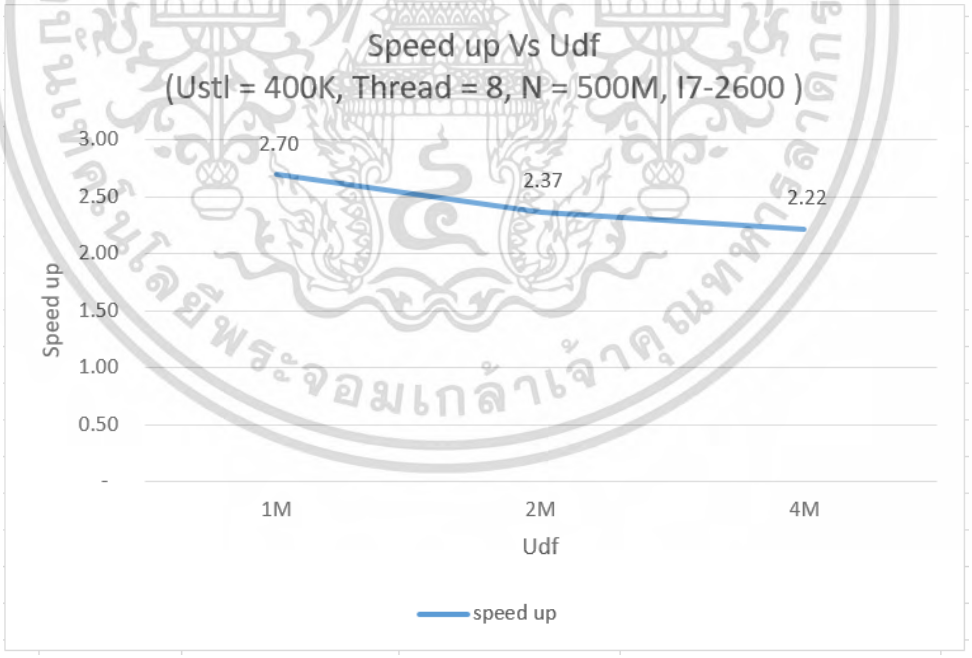
รูปที่ 4.6 เป็นกราฟแสดง Speed up vs $U_{stl}$ บนเครื่อง FX-8320 ซึ่งจากผลการทดลองจะเห็นว่าเมื่อทำการเพิ่มขนาด $U_{stl}$ ให้มีขนาดใหญ่ขึ้นแล้วจะทำให้ Speed up มีค่าสูงขึ้นตามไปด้วย เช่นเดียวกับ รูปที่ 4.7 ที่เป็นกราฟแสดง Speed up vs $U_{stl}$ บนเครื่อง I7-2600 ที่เมื่อทำการ
เพิ่ม $U_{stl}$ ให้มากขึ้นค่า Speed up มีแนวโน้มที่จะมีค่าสูงขึ้นเช่นเดียวกัน ซึ่งเกิดจากการจัดเรียงข้อมูลที่ขนาดชุดข้อมูลที่มีขนาดไม่ใหญ่มากนั้นการจัดเรียงแบบธรรมดานั้นทำงานได้เร็วกว่าแบบขนาน เนื่องจากไม่ต้องทำการสร้าง Thread และ Synchronization

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

4.3.5 Speedup $S$ vs $U_{df}$




**รูปที่ 4.8** กราฟแสดง Speed up vs $U_{df}$ โดยใช้ N = 500M, $U_{stl}$ = 800K, $\tau$ = 8 บนเครื่อง FX-8320




**รูปที่ 4.9** กราฟแสดง Speed up vs $U_{df}$ โดยใช้ N = 500M, $U_{stl}$ = 400K, $\tau$ = 8 บนเครื่อง I7-2600

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

รูปที่ 4.8 เป็นกราฟแสดง Speed up vs $\mathbf{U_{df}}$ บนเครื่อง FX-8320 ซึ่งจากผลการทดลองจะเห็นว่าเมื่อทำการเพิ่มขนาดของ $\mathbf{U_{df}}$ ให้มีขนาดมากขึ้นนั้นทำให้ค่า Speed up มีค่าลดลง เช่นเดียวกับ รูปที่ 4.9 ที่เป็นกราฟแสดง Speed up vs $\mathbf{U_{df}}$ บนเครื่อง I7-2600 ที่เมื่อทำการเพิ่ม $\mathbf{U_{df}}$ ให้มีขนาดมากขึ้นแล้วจะทำให้ Speed up มีค่าลดลงเช่นเดียวกัน ซึ่งพอจะสรุปได้ว่า PDPSort นั้นเมื่อทำการเพิ่มค่า $\mathbf{U_{df}}$ ให้มีขนาดมากขึ้นแล้ว Speed up ลดลง ซึ่งเกิดจากเมื่อขนาด $\mathbf{U_{df}}$ มีขนาดใหญ่นั้นทำให้การเรียกใช้ Depth first schedule เร็วขึ้นทำให้สร้าง Task เพื่อใช้งาน Thread ได้จำนวนน้อยลงทำให้เกิด Thread ว่างงานทำให้ประสิทธิภาพการทำงานลดลง

## 4.4 สรุปผลการทดลอง

ประสิทธิภาพของ PDPSort เมื่อเทียบกับ STLSort แล้ว Speedup และ CPU Utilization นั้น PDPSort มี Maximum speedup อยู่ที่ 6.13 ซึ่งเร็วกว่า HDPSort เพราะในขั้นตอนการแบ่งอาเรย์ตรงกลางของ PDPSort นั้นใช้ Lomuto partition ซึ่งทำการเริ่มสลับตรงจุดกึ่งกลางทำให้ลดจำนวนการเปรียบเทียบลงได้จำนวนหนึ่ง อีกทั้งอาเรย์ทางด้านซ้ายที่ทำการจัดเสร็จแล้วสามารถใช้ OpenMP Task ทำงานไปพร้อมกับการจัดการแบ่งอาเรย์ที่เหลือได้

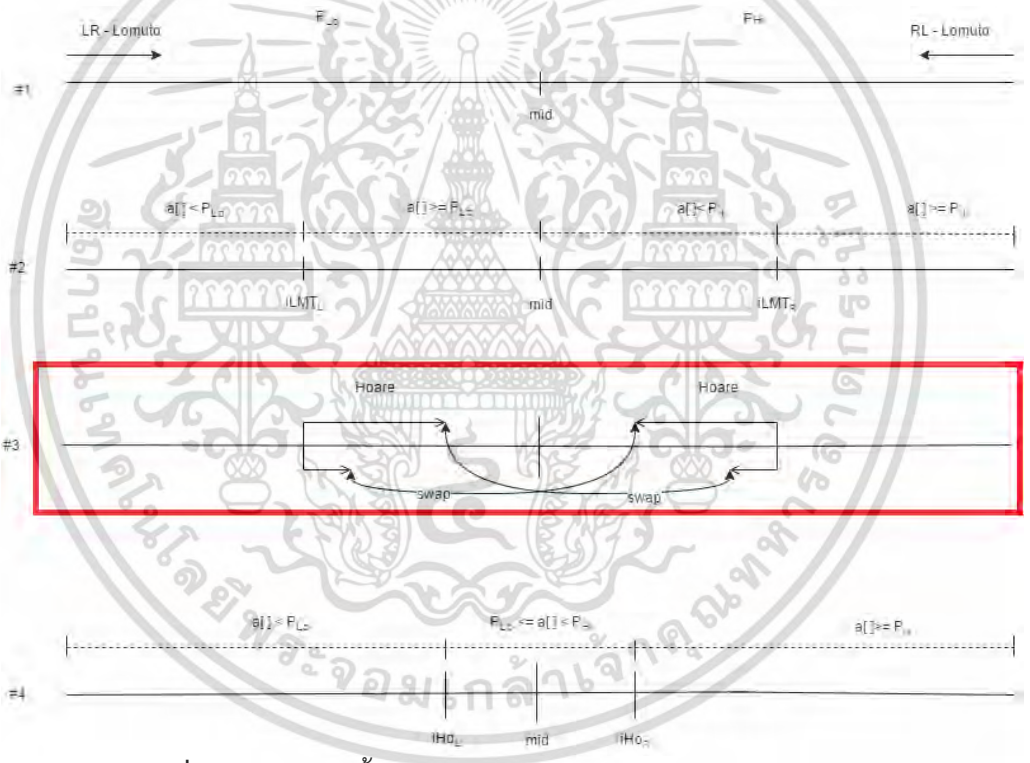
Speed up ของ PDPSort นั้นมีแนวโน้มเพิ่มขึ้นเมื่อทำการใช้ CPU Utilization $\mathbf{V_{cpu}}$ ได้เต็มประสิทธิภาพดูได้จากรูปที่ 4.4 ที่เมื่อ CPU Utilization $\mathbf{V_{cpu}}$ มีค่าสูงขึ้น Speed up ก็เพิ่มขึ้น

จำนวน Hardware thread นั้นมีผลต่อความเร็วในการจัดเรียงของ PDPSort โดยดูได้จากรูปที่ 4.5 ซึ่งเป็นผลมาจากการที่มี Hardware thread จำกัดต่อให้ใช้ Software thread ที่มีจำนวนมากกว่าก็ไม่สามารถเร็วขึ้นมากกว่าเดิม อีกทั้งยังทำให้ทำงานได้ช้าลงอีกเนื่องจากมีการแย่งกันใช้งาน Cache ของแต่ละ Thread

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

# บทที่ 5
# สรุปและสิ่งที่จะพัฒนาต่อไป

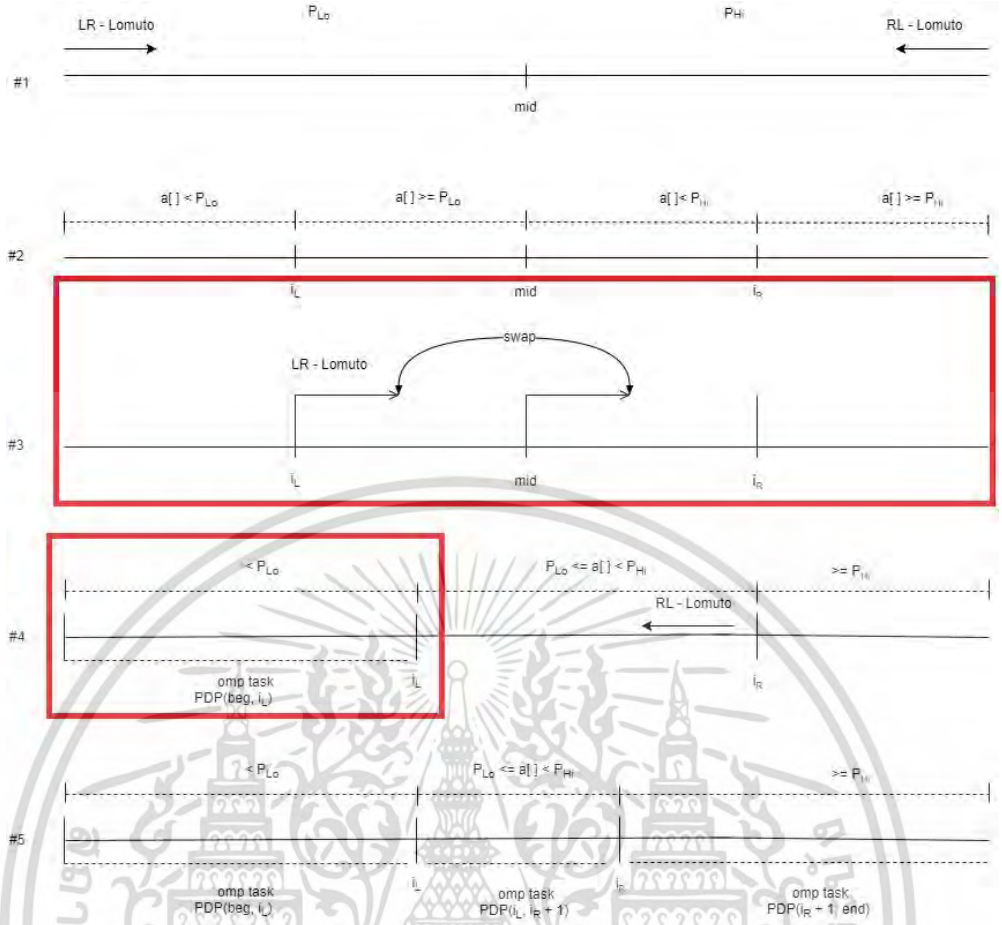
HDPSort และ PDPSort เป็นอัลกอริทึมที่ทำงานแบบขนานเพื่อเพิ่มความเร็วในการจัดเรียง โดยทั้ง HDPSort และ PDPSort นั้นถูกพัฒนาด้วยภาษา C++ และใช้ OpenMP API ทำให้อัลกอริทึมทั้งสองสามารถทำงานแบบขนานได้ โดย HDPSort และ PDPSort นั้นจัดเป็นควิกซอร์ตแบบไพวอทคู่ โดยจากการทดลองบนเครื่อง 4-คอร์ HyperThread Intel i7-2600, 8-core AMD FX-8320 และ 16-thread AMD R7-1700 บนระบบปฏิบัติการ Linux นั้นพบว่า PDPSort สามารถทำงานได้เร็วกว่าแสตนดาร์ดเทมเพลทไลบรารีซอร์ตสูงสุด 2.99, 3.54 และ 6.13 เท่าตามลำดับ




**รูปที่ 5.1** แสดงการขั้นตอนการจัดการอาเรย์ตรงกลางของ HDPSort

HDPSort และ PDPSort นั้นมีขั้นตอนการทำงานเหมือนกันตั้งแต่การทำการเลือกไพวอท, การแบ่งและการ Schedule แต่แตกต่างกันที่ HDPSort ใช้การจัดการแบ่งอาเรย์ตรงกลางด้วย Hoare partition ดังรูปที่ 5.1

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้




**รูปที่ 5.2** แสดงขั้นตอนการจัดการแบ่งอาเรย์ตรงกลางของ PDPSort

แต่ PDPSort ใช้อัลกอริทึม Lomuto ในการจัดการแบ่งตรงกลางดังรูปที่ 5.2 ซึ่งนั่นทำให้ PDPSort ทำงานได้เร็วกว่า HDPSort เนื่องจาก PDPSort นั้นสามารถจัดการแบ่งตรงกลางโดยใช้จำนวนการเปรียบเทียบและสลับน้อยกว่า HDPSort อีกทั้งยังสามารถทำการเรียกใช้ PDPSort ได้แบบ Parallel หลังจากทำการจัดการแบ่งด้านซ้ายเสร็จ

เนื่องจากทั้ง HDPSort และ PDPSort นั้นไม่สามารถทำการจัดการแบ่งตรงกลางได้แบบขนานได้ ดังนั้นสิ่งที่จะพัฒนาต่อไปนั้นคือการพัฒนาให้อัลกอริทึมตัวต่อไปนั้นสามารถทำการจัดการแบ่งตรงกลางได้เพื่อเพิ่มความเร็วในการทำการจัดเรียงขึ้นไปอีก

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

# เอกสารอ้างอิง

[1] V. Yaroslavskiy, “Dual-pivot quicksort algorithm,”2009. [Online]. Available: http://codeblab.com/wpcontent/uploads/2009/09/DualPivotQuicksort.pdf

[2] S. F. Solehria and S. Jadoon, “Multiple pivot sort algorithm is faster than quick sort algorithms: An empirical study,” International Journal of Electrical & Computer Sciences, vol. 11, no. 3, pp. 14–18, 2011.

[3] S. Kushagra, A. Lopez-Ortiz, J. I. Munro, and A. Qiao, “Multi-pivot ´ quicksort: Theory and experiments,” in Proceedings of the Meeting on Algorithm Engineering & Expermiments. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2014, pp. 47–60. [Online]. Available: http://dl.acm.org/citation.cfm?id=2790174.2790180

[4] V. M. Weaver, “Linux perf event features and overhead,” in Second International Workshop on Performance Analysis of Workload Optimized Systems (FastPath 2013), Austin, TX, USA, April 21 2013.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

# ภาคผนวก

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

# ภาคผนวก ก.
# งานวิจัย Parallel Hybrid Dual Pivot Sorting Algorithm

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

2017 14th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)

# Parallel Hybrid Dual Pivot Sorting Algorithm

Surapong Taotiamton
Dept. of Computer Eng., Faculty of Engineering
King Mongkuts Institute of Technology Ladkrabang
Bangkok, Thailand
Email: surapong.taotiamton@gmail.com

Surin Kittitornkun
Dept. of Computer Eng., Faculty of Engineering
King Mongkuts Institute of Technology Ladkrabang
Bangkok, Thailand
Email: surin.ki@kmitl.ac.th

***Abstract*—Sorting is one of the common problems in Computer Science and data analytics. This paper presents empirical results of parallel Hybrid Dual Pivot Sort** ($HDPSort$) **for multicore/manycore CPU systems.** $HDPSort$ **makes use of both classic Lomuto and Hoare partioning algorithms with two pivot values in parallel. It is developed in C++ with OpenMP 3.0 or better.** $HDPSort$ **is benchmarked with the sequential** $STLSort$ **in terms of run time, instruction count and branch load. The Speedups of** $HDPSort$ **are up to 3.02× and 2.79× faster than the** $STLSort$ **on 8-core AMD FX-8320 and 4-core Intel i7-2600 Linux systems, respectively. An indepth analysis shows that** $HDPSort$ **gains the Speedup by 300% over** $STLSort$ **at the expense of 1%-4% of branch mispredictions.**

Keywords-Hoare, Lomuto, OpenMP, Multicore, STL Sort, Partition

## I. Introduction

The most basic computing problem is to sort a very large number of data as fast as possible. It has become very significant for scientific, large-scale biological, social-network applications and so on. Among all sorting algorithms, Divide and Conquer (D&Q) principle is applicable to handle this problem effectively. Examples of D&Q Sorting algorithms are QuickSort [1], [2], MergeSort, etc.

Although QuickSort has been invented since 1962, it is still exciting to enhance parallel D&Q sorting algorithms in terms of performance and efficiency. These challenges are due to single-pivot data partitioning, unbalanced pivot value, and recursive nature of the algorithm. Furthermore, the bottlenecks of parallel D&Q sort should be tackled with dual pivot values along with various hardware characteristics such as intruction counts, branch mispredictions, etc.

In this paper, we have proposed and implemented a parallel Hybrid Dual Pivot Sort ($HDPSort$) for various multicore/manycore CPUs. Our major contributions are summarized as the following.

1) A parallel dual pivot QuickSort with hybrid Lomuto's and Hoare's partition algorithms is proposed and linked with the well-known OpenMP library.
2) The proposed $HDPSort$ algorithm is fully compatible with the Standard Template Library $STLSort$ achieving up to 3.04× faster than $STLSort$ on an 8-core AMD FX-8320 and 4-core Intel i7-2600 Linux machines.

The rest of our paper is organized as follows. Section 2 provides some background and related work to $HDPSort$. Section 3 explains how $HDPSort$ is implemented. The experiment setups and results are presented and discussed in Section 4. Finally, Section 5 concludes and suggests future work.

978-1-4673-9749-0/16/$31.00 ©2016 IEEE

## II. Background and Related Work

### A. Single-Pivot QuickSort Algorithms

QuickSort [1] is the most well-known algorithm based on divide (partition) and conquer concept. The partition algorithm is named after Hoare. Several parallel single-pivot QuickSort based algorithms have been proposed since 1990 by Heidelberger *et al.* [3]. Most of them are based on Hoare's algorithm including PPMQsort [4] and MSTSort [5].

Another partition algorithm is called Lomuto algorithm [6] after Nico Lomuto. Lomuto's is quite cache friendly but weak for worst-case input data. However, it can be applied as a parallel multi-pivot partition algorithm.

### B. Multi-Pivot QuickSort Algorithms

Most of multi-pivot QuickSort algorithms [7], [8], [9] are sequential and in-place algorithms except Mahafzah's [10] which is parallel but needs extra space. Yaroslavskiy [7] introduced and implemented a dual-pivot QuickSort in Java in 2009. The partitioning algorithm was based on Lomuto's. He also roughly estimated the number of comparisons and swaps to be multiple of $n \ln n$ where $n$ is the number of data to be sorted. Solehria and Jadoon [8] presented empirical results of their Dual Pivot sorting with random Integers and Strings. The results include number of operations: comparisons and moves and sorting time. In 2013, Mahafzah [10] splitted the input array with multi-pivot/threads into partitions using extra space and then sort them in parallel with 8 software threads. One year later, Kushagra et al. [9] tried and compared their 3-pivot QuickSort with the single-pivot classic one. They reported that the 3-pivot algorithm is better than the classic one while incurring lower number of comparisons and cache misses.

### C. Standard Template Library Sort (STLSort)

The Standard Template Library $(STL)Sort$ is an outstandingly useful function for sorting any data types with a user-defined comparison function. It is implemented in C++ and also provided as a built-in function for several C++ compilers. Its function prototype is declared in $< algorithm >$ directive as follows.

*void sort(RandomAccessIterator first, RandomAccessIterator last);*



เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

2017 14th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)

The arguments $first$ and $last$ are pointers to the first and the last positions, respectively.

## III. Hybrid Dual Pivot Sort (HDPSort)

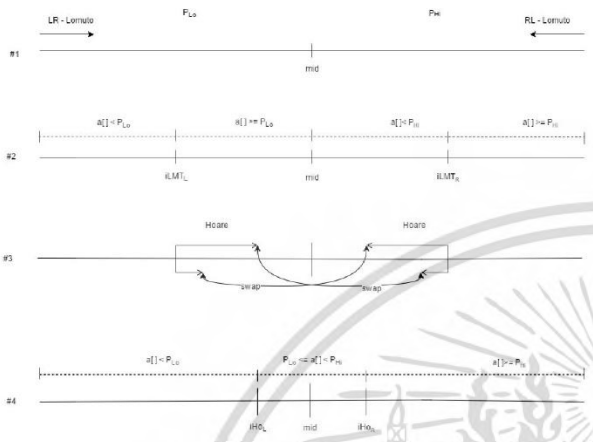



Fig. 1: Basic idea of $HDPSort$ (Note that the figure is not drawn to scale.)

Hybrid Dual Pivot Sort (HDPSort) is a dual pivot partition based QuickSort. Its partition algorithm is hybrid between Lomuto's and Hoare's algorithms. Fig. 1 depicts the concept of HDPSort. Notations used in this paper are listed in Table I.

In line #1 of Fig. 1, two pivot values, $P_{Hi}$ and $P_{Lo}$, are randomly selected. The lower pivot value is applied in Left-Right Lomuto (LR-Lmt) function. On the other hand, the higher pivot value is applied in Right-Left Lomuto (RL-Lmt) partition function. Both functions can be executed in parallel resulting in four partitions, $a[\cdot] < P_{Lo}$, $a[\cdot] \geq P_{Lo}$, $a[\cdot] < P_{Hi}$ and $a[\cdot] \geq P_{Hi}$ as line #2 of Fig. 1.

Both $a[\cdot] \geq P_{Lo}$ and $a[\cdot] < P_{Hi}$ form the undecided middle subarray. The next step (line #3 of Fig. 1) is to apply Hoare's algorithm twice to achieve $P_{Lo} \leq a[\cdot] < P_{Hi}$ as the middle subarray. The resulting three subarrays (line #4 of Fig. 1) shall be divided further recursively until the subarray is finally shorter than $U_{stl}$ elements and then sorted by STLSort.

In details, HDPSort can also be expressed as a pseudocode in Algorithm 1 and developed using C++ language. We make use of the OpenMP Task construct as software threads and schedule them with breadth-first (BF) as default and with depth-first (DF) algorithm once the subarrays are shorter than $U_{df}$ elements.

## IV. Experiments and Discussions

### A. Experiment Setup

The empirical results reported in this paper are based on a multicore CPU, AMD FX-8320 Piledriver and Intel i7-2600 sandy Bridge. Table II provides FX-8320 architectural summary. Both $HDPSort$ and $STLSort$ are evaluated on the systems listed in Table II running 64-bit Ubuntu Linux 14.04 kernel 3.13 LTS. $HDPSort$ is compiled with GCC 4.8.4

**ALGORITHM 1:** Pseudocode of $HDPSort$ Algorithm

```
1  Function Main()
2  |  HDPSort(a, 0, N − 1)
3  EndFunction
4  Function HDPSort(a, beg, end)
5  |  if end − beg + 1 < U_stl then
6  |  |  OpenMP Task
7  |  |  STLSort(a, beg, end)     // Call STLSort
8  |  |  return
9  |  end
10 |  mid = (beg + end)/2
11 |  iP_Lo, iP_Hi = Sample(a, beg, end)
12 |  swap(a, iP_Lo, mid − 1),swap(a, iP_Hi, mid + 1)
13 |  iP_Lo = mid − 1, P_Lo = a[iP_Lo]
14 |  iP_Hi = mid + 1, P_Hi = a[iP_Hi]
15 |  OpenMP Task
16 |  iLMT_L=LR_Lmt(a, beg, iP_Lo − 1, P_Lo)
17 |  OpenMP Task
18 |  iLMT_R=RL_Lmt(a, iP_Hi + 1, end, P_Hi)
19 |  OpenMP Taskwait
20 |  swap(a, iLMT_L, iP_Lo),swap(a, iLMT_R, iP_Hi)
21 |  iHo_L=Hoare(a,iLMT_L,iLMT_R,P_Lo)
22 |  iHo_R=Hoare(a,iLMT_L,iLMT_R,P_Hi)
23 |  if end − beg + 1 > U_df then
24 |  |  OpenMP Task
25 |  |  HDPSort(a, beg,iHo_L-1)          // left
26 |  |  OpenMP Task
27 |  |  HDPSort(a,iHo_L,iHo_R-1)        // middle
28 |  |  OpenMP Task
29 |  |  HDPSort(a,iHo_R,end)            // right
30 |  end
31 |  else
32 |  |  HDPSort(a, beg,iHo_L-1)          // left
33 |  |  HDPSort(a,iHo_L,iHo_R-1)        // middle
34 |  |  HDPSort(a,iHo_R,end)             // right
35 |  end
36 EndFunction
37 Function Hoare(a, bb, ee, p)
38 |  i = bb, j = ee
39 |  while i < j do
40 |  |  while a[i] < p do
41 |  |  |  i++
42 |  |  end
43 |  |  while a[j] ≥ p do
44 |  |  |  j − −
45 |  |  end
46 |  |  swap(a, i, j)
47 |  end
48 |  return j
49 EndFunction
50 Function LR_Lmt(a, bb, ee, p)
51 |  for i = j = bb; i ≤ ee; i++ do
52 |  |  if a[i] < p then
53 |  |  |  swap(a, i, j++)
54 |  |  end
55 |  end
56 |  return j
57 EndFunction
58 Function RL_Lmt(a, bb, ee, p)
59 |  for i = j = ee; i ≥ bb; i − − do
60 |  |  if a[i] ≥ p then
61 |  |  |  swap(a, i, j − −)
62 |  |  end
63 |  end
64 |  return j
65 EndFunction
```



เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

2017 14th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)

TABLE I: Notations used in this paper in alphabetical order

| Notation | Description |
|---|---|
| $a[i]$ | Input data array of $n$ elements |
| $c$ | Number of CPU Cores |
| $i, j$ | Left and right loop indices |
| $iHo_L$ | Left Hoare index |
| $iHo_R$ | Right Hoare index |
| $iLMT_L$ | Left Lomuto index |
| $iLMT_R$ | Right Lomuto index |
| $iP_{Lo}, P_{Lo}$ | The low pivot index and value |
| $iP_{Hi}, P_{Hi}$ | The high pivot index and value |
| $k$ | Number of Processor Sockets |
| K | $10^3$ elements |
| M | $10^6$ elements |
| $n$ | Workload or Data Size (elements) |
| $S$ | Speedup |
| $T_{hdp}$ | Run Time of Hybrid Dual Pivot Sort (Seconds) |
| $T_{stl}$ | Run Time of $STLSort$ (Seconds) |
| $t$ | Number of Hardware Threads |
| $U_{stl}$ | $STLSort$ Cutoff size (elements) |
| $U_{df}$ | Depth-First Cutoff size (elements) |

and linked with OpenMP library under *-fopenmp* option. The measurement tool, Perf [11] version 4.2 is invoked by *perf stat -r 5 -e* to profile every algorithm for 5 times.

TABLE II: Specifications of multicore CPUs in our experiment

| System | i7-2600 | FX-8320 |
|---|---|---|
| Machine | Bare Metal | Bare Metal |
| Code Name | Sandy Bridge | Piledriver |
| Clock (GHz) | 3.40 | 3.50 |
| Sockets ($k$) | 1 | 1 |
| Cores ($c$) | 4 | 8 |
| HyperThread ($t_{max}$) | Yes(8) | No(8) |
| RAM (GB) | 32 | 32 |
| Technology | DDR3-1333 | DDR3-1866 |
| L1 I-Cache | 4x32KB 8-way | 4x64KB 2-way |
| L1 D-Cache | 4x32KB 8-way | 8x16KB 4-way |
| L2 Cache | 4x256KB 8-way | 4x2MB 16-way |
| L3 Cache ($\kappa_{LB}$) | 8MB 16-way | 8MB 64-way |

TABLE III: Parameter set of the experiments

| Parameters | Values |
|---|---|
| Data type | Unsigned Int32 |
| Distribution | Random |
| Workload Size $n$(M) | 100, 200, 300, 400, 500 |
| Threads $t$ | 4, 8, 16, 32 |
| $U_{stl}$(K) | 200, 400, 600, 800 |
| $U_{df}$(M) | 1, 2, 4, 6, 8 |
| Optimization | o2 |

## B. Performance Metrics

In order to examine the behaviors of $HDPSort$ on multicore or manycore CPUs, the following metrics should be measured.

*1) Run Time $T$:* Run Time is measured and averaged by 5 times to compare $T_{hdp}$ and and $T_{stl}$ without data file loading and other overheads.

*2) Speedup $S(\times)$:* Speedup is a ratio for measuring the performance between two algorithms. This metric indicates that how many times our $HDPSort$ or any parallel sort can be executed faster than the sequential $STLSort$. Based on the measured $T_{stl}$ and $T_{hdp}$, $S$ can be computed as

$$S = \frac{T_{stl}}{T_{hdp}} \quad (1)$$

where $\times$ denotes *times*.

*3) Instructions $I$:* Perf [11] is a software tool that relies on a number of hardware/software counters to collect statistics of CPU resource usages with minimal overhead. The number of instructions $I$ accounts for dynamic instructions executed by $t$ threads of workload $n$ elements.

*4) Branch Loads $B$ and Branch Load Misses $B'$:* These metrics can evaluate the algorithm whose performance is limited by branch mispredictions (Branch Load Misses) especially parallel QuickSort and others.

## C. Results and Discussions

In order to understand the behaviors of HDPSort, the experiment results can be divided into several aspects as the following subsections.

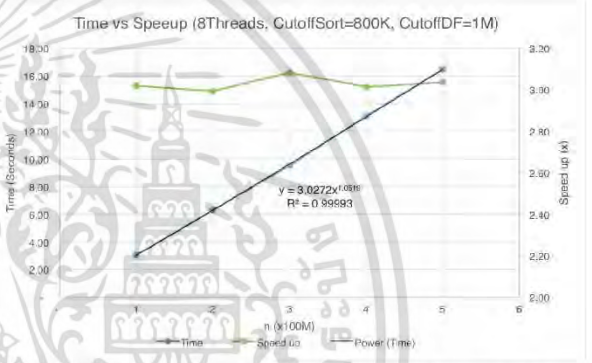



Fig. 2: Best Run Time vs Speedup of $HDPSort$ at different work loads $n$ on FX-8320

*1) Best Run Time vs Speedup:* Fig. 2 plots the best Run Time and its corresponding Speedup of $HDPSort$ on left (blue line) and right (green line) Y axes, respectively. The sizes of work load vary from 100M to 500M. The Speedup is almost constant at 3.00$\times$ faster than STLSort. In addition, it can be observed that the Power regression line (black) of Run Time yield $y = 3.0272x^{1.0519}$ and Determination Coefficient $R^2 = 0.99993$, respectively. This regression equation may correspond to the $n \log n$ run time complexity.

*2) Speedup vs Sorting CutOff $U_{stl}$:* Once the $HDPSort$ partitioning process achieves shorter subarrays. The resulting subarrays that are shorter than $U_{stl}$ elements will be sorted. Fig. 3 depicts Speedup vs Sorting Cutoff on FX-8320 at $n$=400M, $t$=8 threads, $U_{df}$=1M. Given a constant $U_{df}$, Speedup's of $HDPSort$ are getting better as $U_{sort}$ increases.

*3) Speedup vs Scheduling CutOff $U_{df}$:* In Fig. 4 that at $n$=400M, $t$=8 threads, $U_{stl}$=800K and $U_{df}$=1M, as CutoffDF $U_{df}$ increases to 2M, 4M and 8M, Run Time tends to increase. The minimum Run Times are 13.100 and 12.500 Seconds on FX-8320 and i7-2600, respectively. As default, OpenMP makes use of Breadth First scheduling. The DF Cutoff $U_{df}$ allows the algorithm to partition the subarrays further in depth-first fashion. Thus, HDPSort shall be able to exploit cache locality and associativity simultaneously. Speedups of DF scheduling at smaller $U_{df}$ are better than larger $U_{df}$. This can be due to



เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

2017 14th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)

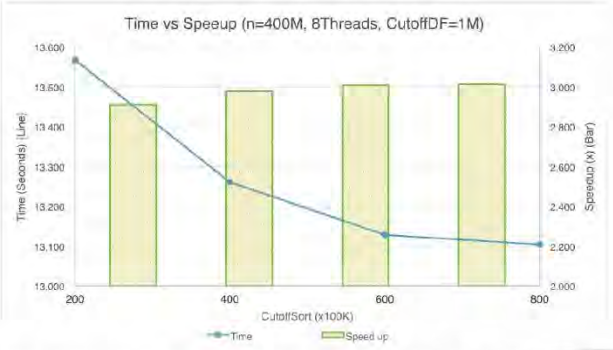



Fig. 3: Speedup vs Sorting Cutoff $U_{stl}$ of HDPSort: $n$=400M, $t$=8 threads, $U_{df}$=1M on FX-8320

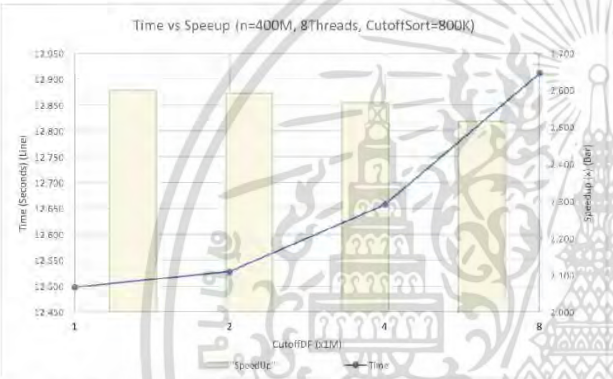



Fig. 4: Speedup vs Depth First Cutoff $U_{df}$ of HDPSort: $n$=400M, $t$=8 threads, $U_{stl}$=800K on i7-2600

the 64-way and 16-way set associative L3 cache of FX-8320 and i7-2600, respectively in Table II.

TABLE IV: Performance comparison on AMD FX-8320, $t = 8$ threads, $U_{stl}$=800k, $U_{df}$=1M

| $n$ | $S(\times)$ | $I_{hdp}/I_{stl}$ | $B_{hdp}/B_{stl}$ | $B'_{hdp}/B'_{stl}$ |
|---|---|---|---|---|
| 100M | 3.02 | 2.93 | 3.22 | 1.02 |
| 200M | 2.99 | 3.00 | 3.29 | 1.04 |
| 300M | 3.08 | 3.03 | 3.34 | 1.02 |
| 400M | 3.01 | 3.10 | 3.40 | 1.02 |
| 500M | 3.04 | 3.08 | 3.40 | 1.01 |

*4) Speedup vs Other Ratios:* Table IV and V summarize the Speedup, Instruction Ratio $I_{hdp}/I_{stl}$, Branch Loads Ratio $B_{hdp}/B_{stl}$ and Branch Load Misses Ratio $B'_{hdp}/B'_{stl}$ of various workloads $n$ on FX-8320 and i7-2600, respectively. It can be noticed that Speedup's show similar trend as Instruction and Branch Load Ratios. Both are due to higher parallelisms. However, the $B'_{hdp}/B'_{stl}$ is just about 1.00 on i7-2600. $HDPSort$ gains the Speedup by 300% over $STLSort$ at the expense of 1%-4% of branch mispredictions on FX-8320. That means the small overhead of branch mispredictions can be traded off with much greater parallelisms on both machines.

TABLE V: Performance comparison on Intel i7-2600, $t = 8$ threads, $U_{stl}$=800k, $U_{df}$=1M

| $n$ | $S(\times)$ | $I_{hdp}/I_{stl}$ | $B_{hdp}/B_{stl}$ | $B'_{hdp}/B'_{stl}$ |
|---|---|---|---|---|
| 100M | 2.62 | 3.10 | 3.32 | 0.99 |
| 200M | 2.54 | 3.12 | 3.31 | 1.01 |
| 300M | 2.79 | 3.14 | 3.36 | 1.01 |
| 400M | 2.59 | 3.21 | 3.41 | 1.00 |
| 500M | 2.54 | 3.23 | 3.43 | 1.00 |

## V. Conclusion

HDPSort is a dual-pivot sorting algorithm whose partition algorithm is hybrid between parallel Lomuto's and sequential Hoare's. Two pivots, $P_{Lo}$ and $P_{Hi}$, are randomly selected for both Left-Right Lomuto and Right-Left Lomuto partition functions resulting in the undecided middle subarray. Then, the classic Hoare's partition is applied twice to cut the middle subarray apart. The resulting three subarrays can be divided recursively until the subarray is finally sorted by a single-pivot STLSort. HDPSort is compared against $STLSort$ in terms of Speedup, Instruction Count, Branch Load and Branch Load Miss. As a parallel algorithm, the achievable Speedup of 3.0× seems to be as good as the number of paritions.

For future work, HDPSort should be fully parallelized further and enhanced to support more pivots.

## Acknowledgment

The authors would like to thank Mr. Apisit Rattanatranurak and Dr. Ratthaslip Ranokpanuwat for their supports.

## References

[1] C. A. R. Hoare, "Quicksort," *ACM*, vol. 4, p. 321, 1962.

[2] R. Sedgewick, "Implementing quicksort program," *Communications of the ACM*, vol. 21, no. 10, pp. 847–857, October 1978.

[3] P. Heidelberger, A. Norton, and J. T. Robinson, "Parallel quicksort using fetch-and-add," *IEEE Transactions on Computers*, vol. 39, no. 1, pp. 847–857, January 1990.

[4] R. Ranokpanuwat and S. Kittitornkun, "Parallel partition and merge quicksort (ppmqsort) on multicore cpus," *J of Supercomputing*, vol. 72, no. 3, pp. 1063–1091, 2016.

[5] A. Rattanatranurak and S. Kittitornkun, "Multi-stack/thread sort: An in-memory parallel sorting algorithm," *submitted for review to IEEE Transactions on Parallel and Distributed Systems*, p. 17, 2017.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[7] V. Yaroslavskiy, "Dual-pivot quicksort algorithm," 2009. [Online]. Available: http://codeblab.com/wp-content/uploads/2009/09/DualPivotQuicksort.pdf

[8] S. F. Solehria and S. Jadoon, "Multiple pivot sort algorithm is faster than quick sort algorithms: An empirical study," *International Journal of Electrical & Computer Sciences*, vol. 11, no. 3, pp. 14–18, 2011.

[9] S. Kushagra, A. López-Ortiz, J. I. Munro, and A. Qiao, "Multi-pivot quicksort: Theory and experiments," in *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2014, pp. 47–60. [Online]. Available: http://dl.acm.org/citation.cfm?id=2790174.2790180

[10] B. A. Mahafzah, "Performance assessment of multithreaded quicksort algorithm on simultaneous multithreaded architecture," *J of Supercomputing*, vol. 66, no. 1, pp. 339–363, 2013.

[11] V. M. Weaver, "Linux perf event features and overhead," in *Second International Workshop on Performance Analysis of Workload Optimized Systems (FastPath 2013)*, Austin, TX, USA, April 21 2013.



เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

# ภาคผนวก ข.

## งานวิจัย A Parallel Dual-Pivot QuickSort Algorithm with Lomuto Partition

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

# A Parallel Dual-Pivot QuickSort Algorithm with Lomuto Partition

Surapong Taotiamton
Dept. of Computer Eng., Faculty of Engineering
King Mongkuts Institute of Technology Ladkrabang
Bangkok, Thailand
Email: surapong.taotiamton@gmail.com

Surin Kittitornkun
Dept. of Computer Eng., Faculty of Engineering
King Mongkuts Institute of Technology Ladkrabang
Bangkok, Thailand
Email: surin.ki@kmitl.ac.th

***Abstract*—Sorting is one of the basic problems in computer Science and big data analytics. This paper presents a parallel Dual Pivot QuickSort ($PDPSort$) for manycore CPU systems. $PDPSort$ makes use of the classic Lomuto partioning algorithm with two pivot values in parallel. It is developed in C++ and linked with OpenMP 4.5. $PDPSort$ is compatible with the Standard Template Library sort (STLSort). The comparison includes run time, Speedup over STLSort and CPU utilization. $PDPSort$ is faster than STLSort by 6.13×, 3.54× and 2.99× on an 8-core 16-thread AMD R7-1700, an 8-core 8-thread AMD FX-8320 and a 4-core Intel i7-2600 Linux systems, respectively.**

Keywords-Lomuto, multi-pivot, QuickSort, Partition, OpenMP

## I. Introduction

The most fundamental computing problem is sorting a number of data as fast as possible. The applications include large-scale biological, scientific, social-network and many others. Among all sorting algorithms, STLSort and its parallel modes are applicable to handle this problem effectively. All of them are based on the original QuickSort algorithms [1], [2].

Although QuickSort has been invented since 1962, only single pivot value is recognized. Several multi-pivot QuickSort algorithms have been proposed [3] since 2009. Most of them work in sequential mode.

In this paper, we have implemented a parallel Dual Pivot STLSort ($PDPSort$) as an alternative on manycore CPUs. Our major contributions are summarized as the following.

1) A multithreaded dual pivot STLSort is proposed and based on a simple yet powerful Lomuto partition algorithm.
2) The proposed $PDPSort$ algorithm is fully compatible with the Standard Template Library STLSort achieving up to 6.13× faster than STLSort on an 8-core 16-thread AMD R7-1700.

Our paper is organized as the following. Section 2 provides some background of multi-pivot QuickSort and related work to $PDPSort$. Section 3 explains how $PDPSort$ is developed. Experiment results are presented and discussed in Section 4. Eventually, Section 5 concludes and suggests future work.

978-1-5386-0787-9/17/$31.00 ©2017 IEEE

## II. Background and Related Work

This section reviews some multi-pivot QuickSort algorithms as well as the STLSort and its parallel functions.

### A. Multi-Pivot QuickSort Algorithms

QuickSort [1] is based on the so-called Hoare's partition algorithm. Another simple yet effective partition algorithm is named Lomuto algorithm [4] after Nico Lomuto. Lomuto's is quite cache friendly and can be applied to partition the data array from left to right and right to left independently in two threads. As a result, it can be applied as a parallel dual-pivot QuickSort algorithm.

Most of multi-pivot QuickSort algorithms [3], [5], [6] are sequential and in-place algorithms except Mahafzah's [7] which is parallel but needs extra space. In 2009, Yaroslavskiy [3] introduced and implemented a dual-pivot QuickSort in Java. The partitioning algorithm is based on Lomuto's. He also roughly estimated the number of comparisons and swaps to be multiple of $n \ln n$ where $n$ is the number of data to be sorted. Solehria and Jadoon [5] presented empirical results of their Dual Pivot sorting with random Integers and Strings in 2011. Their results include number of comparisons and moves as well as sorting time. Mahafzah [7] splitted the input array with multi-pivot/threads into partitions with extra space and then sort them in parallel with 8 software threads in 2013. A year after, Kushagra et al. [6] reported that their 3-pivot algorithm is better than the single-pivot one while incurring lower number of comparisons and cache misses.

### B. Standard Template Library Sort (STLSort)

The Standard Template Library $(STL)Sort$ is a comparison-based sorting function for any data type. The user can define his/her own comparison function. It is provided in C++ language and available in all C++ compilers. Its function prototype is declared in $< algorithm >$ directive as follows.

*void sort(RandomAccessIterator first, RandomAccessIterator last);*

The arguments $first$ and $last$ are pointers to the first and the last positions, respectively. The GNU libstdc++ parallel mode, namely Balanced QuickSort[8] is a parallel QuickSort function with single-pivot partition algorithm.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

TABLE I: Notations used in this paper in alphabetical order

| Notation | Description |
|---|---|
| $a[i]$ | Input data array of $N$ elements |
| $c$ | Number of CPU Cores |
| $i, j$ | Left and right loop indices |
| $i_L$ | Left Lomuto index |
| $i_R$ | Right Lomuto index |
| $iP_{Lo}, P_{Lo}$ | The low pivot index and value |
| $iP_{Hi}, P_{Hi}$ | The high pivot index and value |
| $k$ | Number of Processor Sockets |
| K | $10^3$ elements |
| M | $10^6$ elements |
| $N$ | Workload or Data Size (elements) |
| $S$ | Speedup |
| $T_{pdp}$ | Run Time of Dual Pivot QuickSort (Seconds) |
| $T_{stl}$ | Run Time of STLSort (Seconds) |
| $\tau$ | Number of Hardware Threads |
| $\tau_{max}$ | Maximum Hardware Threads |
| $U_{stl}$ | STLSort Cutoff size (elements) |
| $U_{df}$ | Depth-First Cutoff size (elements) |
| $V_{cpu}$ | CPU Utilization (non-Idle) $\leq 1.00$ |

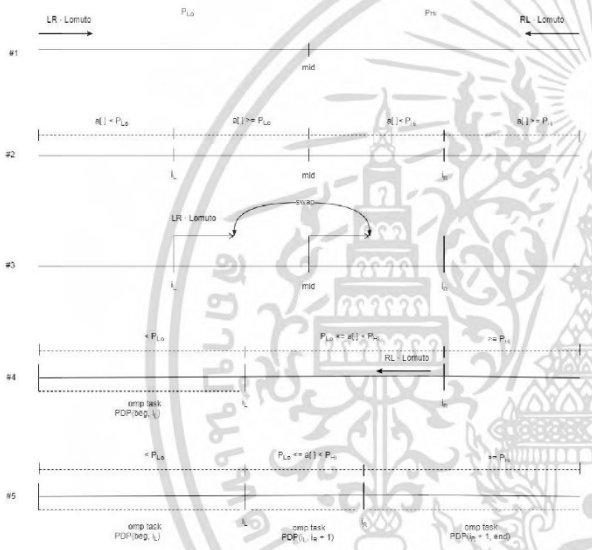



Fig. 1: Basic concept of $PDPSort$ (Note that the figure is not drawn to scale.)

## III. Parallel Dual Pivot QuickSort (PDPSort)

The proposed PDPSort's dual pivot partition is based on Lomuto's partition only due to its simplicity and cache friendliness. Fig. 1 shows the concept of PDPSort. PDPSort is similar to HDPSort in [9]. However, we remove Hoare's partition because it is not as flexible as Lomuto's. Further details will be discussed soon. Notations used in this paper are listed in Table I.

Although PDPSort is developed in C++, it can also be pseudocoded as listed in Algorithm 1. Due to its recursive nature, OpenMP Task construct is applied as a software thread. Therefore, two scheduling algorithms of choice are breadth-first (BF) as default and depth-first (DF) once the subarrays are shorter than $U_{df}$ elements.

In line #1 of Fig. 1, two pivot values, $P_{Hi}$ and $P_{Lo}$, are randomly selected. The lower pivot value is applied in Left-Right Lomuto (LR-Lmt) function as an OpenMP task. On the other hand, the higher pivot value is applied in Right-Left Lomuto (RL-Lmt) partition function as well. Both tasks can be executed in parallel resulting in four partitions, $a[\cdot] < P_{Lo}$, $a[\cdot] \geq P_{Lo}$, $a[\cdot] < P_{Hi}$ and $a[\cdot] \geq P_{Hi}$ as line #2 of Fig. 1.

The middle subarrays correspond to $a[\cdot] \geq P_{Lo}$ and $a[\cdot] < P_{Hi}$. The next step (line #3 of Fig. 1) is to apply Left to Right Lomuto's algorithm again from the $mid$ position to achieve the final $a[\cdot] < P_{Lo}$ partition. Once the left-most partition ($a[\cdot] < P_{Lo}$) is finally obtained, the next recursive call is invoked in either line 24 or 27 of Alg. 1. Unlike [9], we remove Hoare's partition algorithm to save the number of operations based on this modified Lomuto's as shown in line 45 of Alg. 1. That is because Hoare's algorithm needs to scan from both sides. Lomuto's can be modified such that it starts scanning from any desired locaiton.

The Right to Left Lomuto's (line 29 of Alg. 1) results in two partitions the middle subarray ($P_{Lo} \leq a[\cdot] < P_{Hi}$) and the right-most one ($a[\cdot] \geq P_{Hi}$ partition). These two subarrays (line #5 of Fig. 1) shall be divided further recursively until the subarray is subsequently shorter than $U_{stl}$ elements and then STLSorted.

## IV. Experiments and Discussions

This section describes how the experiments are set up. Leter on, results are shown and discussed based on key performance metrics.

### A. Experiment Setup

Both $PDPSort$ and STLSort are evaluated on the systems listed in Table III running 64-bit Ubuntu Linux 17.04 kernel 4.10.0-26. $PDPSort$ is compiled with GCC 6.3.4 and linked with OpenMP 4.5 library under *-fopenmp* option. The measurement tool, Perf [10] version 4.10.17 is invoked by *perf stat -r 5 -e* to profile every algorithm for 5 times. Preliminary results reported in this paper are based on the following CPUs, 4-core 8-thread Intel sandy Bridge i7-2600, 8-core 8-thread AMD FX-8320 and 8-core 16-thread AMD Ryzen R7-1700. Table III provides their architectural summary.

### B. Performance Metrics

In order to investigate the characteristics of $PDPSort$ on manycore CPUs, the following metrics should be measured.

*1) CPU Utilization ($V_{cpu}$):* The metric can be obtained from the contents of $/proc/stat$ file which keeps track of statistics of all HyperThread or physical hardware thread. The contents can be divided into columns: *user*, *system*, *idle* and so on. The non-idle CPU Utilization can be computed from

$$V_{cpu} = \frac{1}{t_{max}} \sum_{i=0}^{t_{max}-1} V_{cpu_i} \quad (1)$$

where $t_{max}$ denotes maximum hardware threads and $V_{cpu_i}$ denotes non-idle CPU Utilization of hardware thread $i$. The higher CPU Utilization, the shorter parallel CPU Time is.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

**ALGORITHM 1:** Pseudocode of $PDPSort$ Algorithm

```
1  Function Main()
2  |  PDPSort(a, 0, N − 1)
3  EndFunction
4  Function PDPSort(a, beg, end)
5  |  if end − beg + 1 < U_stl then
6  |  |  OpenMP Task
7  |  |  STLSort(a, beg, end)      // Call STLSort
8  |  |  return
9  |  end
10 |  mid = (beg + end)/2
11 |  iP_Lo, iP_Hi = Sample(a, beg, end)
12 |  swap(a, iP_Lo, mid − 1), swap(a, iP_Hi, mid + 1)
13 |  iP_Lo = mid − 1, P_Lo = a[iP_Lo]
14 |  iP_Hi = mid + 1, P_Hi = a[iP_Hi]
15 |  OpenMP Task
16 |  i_L=LR_Lmt(a, beg, iP_Lo − 1, beg, P_Lo)
17 |  OpenMP Task
18 |  i_R=RL_Lmt(a, iP_Hi + 1, end, end, P_Hi)
19 |  OpenMP Taskwait
20 |  swap(a, i_L, iP_Lo), swap(a, i_R, iP_Hi)
21 |  i_L=LR_Lmt(a, mid, i_R, i_L, P_Lo)
22 |  if i_L − beg + 1 > U_df then
23 |  |  OpenMP Task
24 |  |  PDPSort(a, beg, i_L)          // left
25 |  end
26 |  else
27 |  |  PDPSort(a, beg, i_L)          // left
28 |  end
29 |  i_R=RL_Lmt(a, i_L, i_R, i_R, P_Hi)
30 |  if i_R − i_L > U_df then
31 |  |  OpenMP Task
32 |  |  PDPSort(a, i_L, i_R+1)        // middle
33 |  end
34 |  else
35 |  |  PDPSort(a, i_L, i_R+1)        // middle
36 |  end
37 |  if end − i_R + 1 > U_df then
38 |  |  OpenMP Task
39 |  |  PDPSort(a, i_R+1, end)        // right
40 |  end
41 |  else
42 |  |  PDPSort(a, i_R+1, end)        // right
43 |  end
44 EndFunction
45 Function LR_Lmt(a, bb, ee, j, p)
46 |  for i = bb; i ≤ ee; i++ do
47 |  |  if a[i] < p then
48 |  |  |  swap(a, i, j++)
49 |  |  end
50 |  end
51 |  return j
52 EndFunction
53 Function RL_Lmt(a, bb, ee, j, p)
54 |  for i = ee; i ≥ bb; i − − do
55 |  |  if a[i] ≥ p then
56 |  |  |  swap(a, i, j − −)
57 |  |  end
58 |  end
59 |  return j
60 EndFunction
```

TABLE II: Parameter set of the experiments

| Parameters | Values |
|---|---|
| Data type | Unsigned Int32 |
| Distribution | Random |
| Size $N$(M) | 100,200,300,400,500 |
| $U_{stl}$(K) | 400, 600, 800 |
| $U_{df}$(M) | 1, 2, 4 |
| Optimization | o2 |

TABLE III: Specifications of multicore CPUs in our experiment, SMT: Simultaneous MultiThreading

| System Name | i7-2600 Sandy Bridge | FX-8320 Piledriver | R7-1700 Ryzen |
|---|---|---|---|
| Clock (GHz) | 3.40 | 3.50 | 3.00 |
| Cores ($c$) | 4 | 8 | 8 |
| SMT($t_{max}$) | Yes(8) | No(8) | Yes(16) |
| RAM (GB) | 32 | 32 | 32 |
| Technology | DDR3-1333 | DDR3-1866 | DDR4-2133 |
| L1 I-Cache | 4x32KB 8-way | 4x64KB 2-way | 8x64KB 4-way |
| L1 D-Cache | 4x32KB 8-way | 8x16KB 4-way | 8x32KB 8-way |
| L2 Cache | 4x256KB 8-way | 4x2MB 16-way | 8x512MB 8-way |
| L3 Cache | 8MB 64-way | 8MB 64-way | 2x8MB 16-way |

*2) Run Time $T$:* Run Time is measured and averaged by 5 times to compare $T_{pdp}$ and and $T_{stl}$ without data file loading and other overheads.

*3) Speedup $S(\times)$:* Speedup is the key performance indicator computed as a ratio of two algorithms' run times. It indicates how many folds $PDPSort$ outperforms STLSort. Based on the measured $T_{stl}$ and $T_{pdp}$, $S$ can be computed as

$$S = \frac{T_{stl}}{T_{pdp}} \tag{2}$$

where $\times$ denotes *times*.

### C. Results and Discussions

In order to investigate of PDPSort in several aspects, the experiment results can be divided as the following subsections.

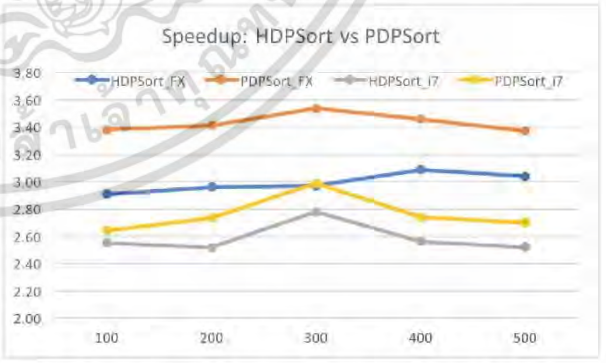



Fig. 2: Speedup: PDPSort vs HDPSort, $N$=100M-500M on FX-8320 and i7-2600

*1) PDPSort vs. HDPSort [9]:* In order to compare PDPSort with its predecessor directly, we share the same dataset with HDPSort [9]. The sizes of work load vary from 100M to 500M. Figure 2 compares the maximum Speedup of $PDPSort$ vs

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

HDPSort on both FX8320 and i7-2600 systems. Speedup's on FX-8320 range from at 3.3× to 3.5× faster than STLSort while those of HDPSort [9] are between 2.9× to 3.1×. Similarly, PDPSort Speedup's on i7-2600 are improved as well.

That is because the middle subarrays are partitioned by Lomuto's again resulting in less run time complexity. Therefore, the left-most subarray can be forked as a Task earlier. Similarly, the middle subarray can be obtained the same way.

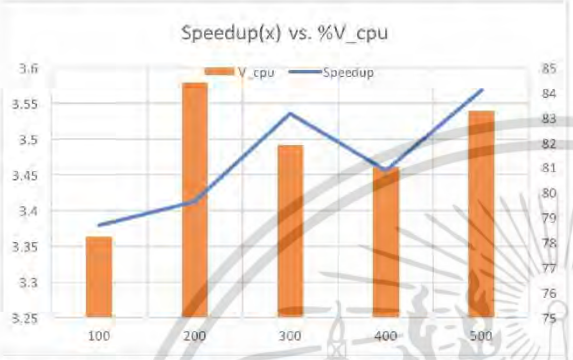



Fig. 3: Speedup vs $\%V_{cpu}$: $N$=100M-500M, $\tau$=16 threads, $U_{df}$=800K, $U_{df}$=1M on FX-8320

*2) Speedup $S$ vs. CPU Utilization $V_{cpu}$:* This subsection is intended to show that PDPSort can exploits the CPU cores well and there is more room to improve even further. In Figure 3, Speedup $S$ and CPU Utilization $V_{cpu}$ of FX8320 are plotted in line and bar graphs, respectively where $\tau$ = 16 threads, $U_{stl}$ = 800K, $U_{df}$ = 1M. It can be observed that as the workload grows the Speedup tends to be higher. The utilization is relatively improved from 78% to 84%. It can be concluded that the higher CPU Utilization $V_{cpu}$, the higher Speedup $S$.

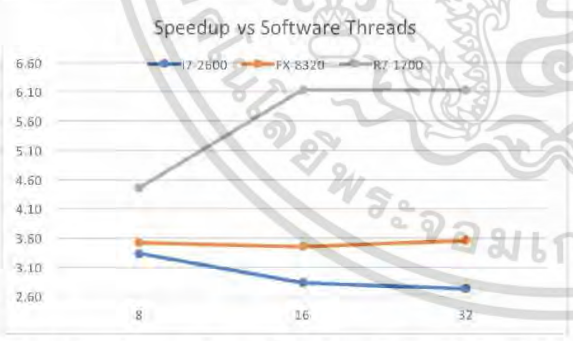



Fig. 4: Speedup vs software Threads $\tau$= 8, 16, 32, $N$=400M

*3) Speedup $S$ vs. Software Threads $\tau$:* Figure 4 illustrates the trends of Speedup versus software thread count on various systems. The Speedup of R7-1700 peaks at $\tau$ = 16 and 32 threads due to its many more (8) cores and 16 hardware threads than others. On the contrary, PDPSort may not handle software threads $\tau > t_{max}$ on i7-2600 well. Based on measurements by Perf [10], more software threads cause even more data TLB store misses and cache misses as they share the limited number of CPU cores and hardware threads. Data TLB store misses can slow the algorithm down due to Page Table accesses and even Page Faults.

## V. Conclusion

PDPSort is a dual-pivot QuickSort whose partition process applies Lomuto's partition algorithm in multithreaded. The array is partitioned with two pivots, $P_{Lo}$ and $P_{Hi}$ from both ends with two threads resulting in two undecided middle subarrays. Then, the Lomuto's is applied twice to achieve three subarrays. These subarrays can be divided individually and recursively until it is finally sorted by a single-pivot STLSort. PDPSort is benchmarked against STLSort in terms of Speedup and CPU utilization. As a parallel algorithm, the achievable maximum Speedup is 6.13× over STLSort on an 8-core 16-thread AMD R7-1700 17.04 Ubuntu Linux system and better than our previous HDPSort algorithm. Our PDPSort shall be improved further to support more pivots and threads to improve CPU utilization as future work.

## Acknowledgment

The authors would like to thank Mr. Apisit Rattanatranurak of Suan Sunandha Rajabhat University and Dr. Ratthaslip Ranokpanuwat of Dhurakij Pundit University for their supports.

## References

[1] C. A. R. Hoare, "Quicksort," *ACM*, vol. 4, p. 321, 1962.

[2] R. Sedgewick, "Implementing quicksort program," *Communications of the ACM*, vol. 21, no. 10, pp. 847–857, October 1978.

[3] V. Yaroslavskiy, "Dual-pivot quicksort algorithm," 2009. [Online]. Available: http://codeblab.com/wp-content/uploads/2009/09/DualPivotQuicksort.pdf

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[5] S. F. Solehria and S. Jadoon, "Multiple pivot sort algorithm is faster than quick sort algorithms: An empirical study," *International Journal of Electrical & Computer Sciences*, vol. 11, no. 3, pp. 14–18, 2011.

[6] S. Kushagra, A. López-Ortiz, J. I. Munro, and A. Qiao, "Multi-pivot quicksort: Theory and experiments," in *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2014, pp. 47–60. [Online]. Available: http://dl.acm.org/citation.cfm?id=2790174.2790180

[7] B. A. Mahafzah, "Performance assessment of multithreaded quicksort algorithm on simultaneous multithreaded architecture," *J of Supercomputing*, vol. 66, no. 1, pp. 339–363, 2013.

[8] J. Singler, P. Sanders, and F. Putze, "Mcstl : The multi-core standard template library," *Euro-Par 2007 Parallel Processing. Springer Berlin Heidelberg*, pp. 682–694, 2007.

[9] S. Taotiamton and S. Kittitornkun, "Parallel hybrid dual pivot sorting algorithm," in *Proceedings of ECTI 2017*, 2017.

[10] V. M. Weaver, "Linux perf event features and overhead," in *Second International Workshop on Performance Analysis of Workload Optimized Systems (FastPath 2013)*, Austin, TX, USA, April 21 2013.

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้

# ประวัติผู้เขียน

| | |
|---|---|
| ชื่อ-นามสกุล | นายสุรพงศ์ เท่าเทียมตน |
| วัน เดือน ปีเกิด | 17 ตุลาคม 2535 ที่สมุทรปราการ |
| ที่อยู่ | 13 หมู่ 7 ถ.สุขุมวิท ต.ท้ายบ้านใหม่ อ.เมือง จ.สมุทรปราการ 10280 โทร.0-2323-4030 |
| ประวัติการศึกษา | 2557 วิศวกรรมศาสตรบัณฑิต สาขาวิชาวิศวกรรมคอมพิวเตอร์ (เกียรตินิยมอันดับ 2) สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง |
| ความชำนาญเฉพาะด้าน | 1.) การพัฒนาแอปพลิเคชัน<br>2.) การเขียนโปรแกรมแบบขนาน |
| ประสบการณ์การทำงานและผลงานวิจัย | |
| พ.ศ.2557 - ปัจจุบัน | ฝึกงานตำแหน่ง Software Developer บริษัท เน็ตเบย์ จำกัด (มหาชน) |

เอกสารนี้เป็นเอกสารที่สงวนไว้สำหรับการใช้งานเพื่อการศึกษาเท่านั้น ไม่อนุญาตให้นำไปใช้ประโยชน์ด้านการค้า ไม่ว่ากรณีใดๆ ทั้งสิ้น อีกทั้งห้ามมิให้ดัดแปลงเนื้อหา และต้องอ้างอิงถึงเจ้าของเอกสารทุกครั้งที่มีการนำไปใช้