กรรมวิธีการประมวลผลแบบขนานบน พีซี คลัสเตอร์

Parallel Processing Method on PC Cluster

ธีระ หล่อตระกูลชัย
วิศิษฎ์ พัฒน์ชู
เสริมศักดิ์ ศรีโพธิ์ทอง

ปัญหาพิเศษนี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรวิทยาศาสตรบัณฑิต
ภาควิชาคณิตศาสตร์และวิทยาการคอมพิวเตอร์
คณะวิทยาศาสตร์
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
ปีการศึกษา 2542

Parallel Processing Method on PC Cluster

Theera Lohtrakulchai

Wisit Phatchoo

Sermsak Sriphothong

A SPECIAL PROJECT SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENT FOR THE DEGREE OF BACHELOR OF SCIENCE

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCES

FACULTY OF SCIENCE

KING MONGKUT'S INSTITUTE OF TECHNOLOGY LARDKRABANG

ACADEMIC YEAR 1999

หัวข้อปัญหาพิเศษ	กรรมวิธีการประมวลผลแบบขนานบน พีซี คลัสเตอร์	
	Parallel Processing Method on PC Cluster	
ชื่อนักศึกษา	นายธีระ หล่อตระกูลชัย	39054623
	นายวิศิษฎ์ พัฒน์ชู	39054662
	นายเสริมศักดิ์ ศรีโพธิ์ทอง	39054679
ภาควิชา	คณิตศาสตร์และวิทยาการคอมพิวเตอร์	
สาขา	วิทยาการคอมพิวเตอร์	
อาจารย์ที่ปรึกษา	อาจารย์ไพรบูลย์ พันธรักษ์พงษ์	

ภาควิชาคณิตศาสตร์และวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง อนุมัติให้นับปัญหาพิเศษฉบับนี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรวิทยาศาสตรบัณฑิต สาขาวิทยาการคอมพิวเตอร์ ประจำปีการศึกษา 2542

คณะกรรมการสอบ		ลายมือชื่อ
ประธานกรรมการ	อาจารย์ธีรวัฒน์ ประกอบผล	
กรรมการ	อาจารย์วิสันต์ ตั้งวงษ์เจริญ	
กรรมการและอาจารย์ที่ปรึกษา	อาจารย์ไพรบูลย์ พันธรักษ์พงษ์	

(อาจารย์ไพรบูลย์ พันธรักษ์พงษ์)
หัวหน้าภาควิชาคณิตศาสตร์และวิทยาการคอมพิวเตอร์

หัวข้อปัญหาพิเศษ	กรรมวิธีการประมวลผลแบบขนานบน พีซี คลัสเตอร์	
ชื่อนักศึกษา	นายธีระ หล่อตระกูลชัย	39054623
	นายวิศิษฎ์ พัฒน์ชู	39054662
	นายเสริมศักดิ์ ศรีโพธิ์ทอง	39054679
ปริญญา	วิทยาศาสตรบัณฑิต	
ภาควิชา	คณิตศาสตร์และวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์	
สาขาวิชา	วิทยาการคอมพิวเตอร์	
ปีการศึกษา	2542	
อาจารย์ที่ปรึกษา	อาจารย์ ไพรบูลย์ พันธรักษ์พงษ์	

บทคัดย่อ

Parallel Processing เป็นวิธีการที่ใช้ในการแก้ปัญหาของการคำนวณขนาดใหญ่ ที่ประกอบด้วยการแตกงาน(task) จำนวนมาก มาแบ่งให้ CPU ช่วยกันประมวลผลงานเหล่านั้น ทำให้สามารถประมวลผลงานที่ใช้เวลาในการประมวลผลมากหรืองานต้องการการคำนวณขนาดใหญ่ ได้เร็วกว่าการประมวลผลแบบตามลำดับ เครื่องคอมพิวเตอร์ทั่วๆไปจะมีการทำงานแบบที่ใช้ CPU ประมวลผลงานหรือโปรแกรมตัวเดียว แต่ถ้าเมื่อเราเอา CPU หลายตัวมาประมวลผลงานหรือโปรแกรมแบบเดียวกันนี้ก็จะทำให้ใช้เวลาในการประมวลผลลดลง ยิ่งใช้ CPU มากเท่าไร เวลาในการประมวลผลยิ่งลดลง แต่การที่จะเอา CPU มาช่วยกันประมวลผลนี้ส่วนใหญ่เครื่องที่มีการทำงานแบบนี้นั้นจะเป็นพวกเครื่องแบบ Multiprocessor หรือเครื่องระดับ Supercomputer ซึ่งเครื่องเหล่านี้มีราคาสูง จะเกิดแนวคิดใหม่ที่เรียกว่า “Cluster Computer” มาเพื่อให้มีการทำงานเหมือนมีการใช้ CPU หลายตัวมาประมวลผลเหมือนที่ใช้ในเครื่อง Multiprocessor

ดังนั้นถ้าเราเอาเทคโนโลยี Cluster Computer กับ Parallel Processing มาทำงานร่วมกันจะทำให้การทำงานงานใดงานหนึ่งหรือนำมาประมวลผลงานจะทำให้สามารถทำงานหรือประมวลผลงานต่างได้อย่างมีประสิทธิภาพและใช้เวลาในการประมวลเร็วโดยที่ไม่จำเป็นต้องไปใช้เครื่อง Supercomputer ที่มีราคาแพง หรือเครื่องที่มีความเร็วในการประมวลผลสูงแต่มีราคาแพง

Special Project Tittle	Parallel Processing Method on PC Cluster	
Students	Mr. Theera Lohtrakulchai	39054623
	Mr. Wisit Phatchoo	39054662
	Mr. Sermsak Sriphothong	39054679
Degree	Bachelor's Degree of Science	
Department	Mathematics and Computer Sciences, Faculty of Science	
Programme	Computer Sciences	
Academic Year	1999	
Special Project Advistor	Lecturer Praiboon Pantaragphong	

Abstract

Parallel Processing is the solution method for large problem that compose with many small tasks. CPU will compute that small task. This reason, time is spend for computation of large problem or very big task is few than computation with sequential processing. Most computer compute task or program with one CPU. If we use several CPU compute small task, time is spend for computation will reduced. If we increase CPU, time is spend for computation will more reduced. This theory is used in Multiprocessor computer or Supercomputer. The Multiprocessor or Supercomputer is expensive. It will appear new idea, it is "Cluster Computer" to processing program like Multiprocessor.

If we will use both idea, Parallel Processing and Cluster Computer, for compute program. It will compute task or program efficiently and time is spend for computation will more reduced. Both idea is one way for who want the efficient computation but it use few cost than Multiprocessor or expensive high performance computer.

กิตติกรรมประกาศ

ในการทำปัญหาพิเศษเรื่องกรรมวิธีการประมวลผลแบบขนานบน PC Cluster สำเร็จไปได้ด้วยดี คณะผู้จัดทำต้องขอขอบพระคุณ อาจารย์ไพรบูลย์ พันธรักษ์พงษ์ อาจารย์ที่ปรึกษาปัญหาพิเศษฉบับนี้ ที่กรุณาให้คำแนะนำและวางแนวทางในการแก้ปัญหาต่างๆ และยังเป็นผู้ตรวจสอบความถูกต้องให้เป็นไปตามจุดประสงค์ที่ได้ตั้งไว้

และยังต้องขอขอบคุณเพื่อนๆ ที่ให้การช่วยเหลือ ทีมงานผู้สร้างซอฟแวร์ PVM ที่ให้ความช่วยเหลือในการเขียนโปรแกรม รวมทั้งพ่อแม่ที่สนับสนุนด้านเงินทุนและพ่อแม่ของเพื่อนที่ให้ความอนุเคราะห์ด้านสถานที่ทำปัญหาพิเศษนี้ ตลอดจนทุกคนที่มีส่วนเกี่ยวข้องกับปัญหาพิเศษไว้ ณ. โอกาสนี้ด้วย

คณะผู้จัดทำ
มีนาคม 2543

สารบัญ

หน้า

หน้า

หน้า

สารบัญภาพ

รูปที่ **หน้า**

รูปที่ **หน้า**

สารบัญตาราง

ตารางที่ **หน้า**

บทที่ 1
บทนำ

1.1 ความเป็นมาและความสำคัญของปัญหา

ในปัจจุบัน Personal Computer ราคาถูกลงมากเมื่อเทียบกับสมัยก่อน ในขณะที่มีความสามารถในการทำงานเพิ่มสูงขึ้นแต่ก็ยังไม่เพียงพอต่อความต้องการที่จะประมวลผลงานขนาดใหญ่ จึงเกิดความคิดที่จะนำ PC หลายๆเครื่องมาต่อ network กันแล้วนำมาช่วยกันประมวลผลงานได้ เพื่อที่จะใช้ในการประมวลผลงานขนาดใหญ่ได้เร็วขึ้น โดยที่ไม่จำเป็นจะต้องใช้ Supercomputer ซึ่งมีราคาแพงกว่าเครื่อง PC มาก

การทำงานของโปรแกรมแบบตามลำดับนั้นมีการใช้งานกันแพร่หลาย โปรแกรมส่วนใหญ่ก็มีการทำงานแบบนี้ แต่สำหรับการทำงานของโปรแกรมที่มีขนาดใหญ่ถ้าการทำงานแบบตามลำดับจะทำให้ทำงานได้ช้าและไม่มีประสิทธิภาพ จึงมีแนวคิดที่จะทำงานแบบขนานคือ การทำงานของโปรแกรมจะแตกเป็น Process ย่อยและนำไปให้ CPU ประมวลผล process ย่อยๆเหล่านั้น ดังนั้น ถ้านำเอา 2 แนวคิดของการนำเอา PC มาต่อ network กันเพื่อช่วยกันประมวลผลกับแนวคิดการประมวลผลแบบขนาน จะทำให้การทำงานของโปรแกรมมีการทำงานที่มีประสิทธิภาพมากขึ้นและใช้เวลาในการประมวลผลน้อยลง

1.2 ความมุ่งหมายและวัตถุประสงค์ของการศึกษา

1 เพื่อที่จะศึกษาการนำเครื่องไมโครคอมพิวเตอร์(PC) ซึ่งมีหน่วยประมวลผลกลาง(CPU) เครื่องละหนึ่งหน่วยและแต่ละเครื่องมีหน่วยความจำ เป็นของตนเอง มาประยุกต์ใช้งานร่วมกัน ช่วยกันประมวลผล

2 เพื่อศึกษาการเขียนโปรแกรมแบบขนาน สามารถที่จะเขียนโปรแกรมตามหลักการของการประมวลผลแบบขนานได้

3 สามารถที่จะแสดงให้เห็นว่าการประมวลผลแบบขนานจะทำให้สามารถประมวลผลงานที่มีขนาดใหญ่ได้เร็วกว่าที่จะทำการประมวลผลแบบลำดับ(sequential)

1.3 สมมุติฐานของการศึกษา

สามารถนำไมโครคอมพิวเตอร์มากกว่า 1 เครื่องมาช่วยกันประมวลผลงานขนาดใหญ่เพื่อที่จะทำให้สามารถประมวลผลงานนั้นได้เร็วขึ้นกว่าการที่ใช้คอมพิวเตอรีเครื่องเดียวประมวลผล

1.4 ขอบเขตการศึกษา

1. นำ PC มาทำ cluster ได้
2. เปรียบเทียบการประมวลผลแบบขนาน กับ แบบลำดับได้

1.5 ขั้นตอนของการศึกษา

1 ศึกษาสถาปัตยกรรมของการทำ cluster บน PC เพื่อสนับสนุนการประมวลผลแบบขนาน

2 ศึกษาหาวิธีการหรือไลบรารี่ใดที่จะทำให้สามารถนำ PC มาต่อโดยผ่าน network แล้วสามารถให้ PC มาร่วมกันประมวลผลงานได้

3 ติดตั้งไลบรารี่ที่ทำให้ PC สามารถทำงานร่วมกันประมวลผล

4 ศึกษาการทำงานและพื้นฐานการเขียนโปรแกรมแบบขนาน

5 ศึกษาการส่งและรับ message และ process ระหว่างเครื่อง PC

6 ศึกษา algorithm ตัวอย่างในการทำงานแบบขนาน

7 ศึกษาและยกตัวอย่างโปรแกรมที่ทำงานแบบขนานแล้วนำมาเปรียบเทียบการทำงานแบบลำดับ

1.6 ข้อตกลงเบื้องต้น

ต้องมีการต่อคอมพิวเตอร์ผ่านระบบเครือค่าย LAN ตั้งแต่ 2 เครื่องขึ้นไป ต้องใช้ระบบปฏิบัติการ LINUX และต้องมี library PVM install และ config ได้ถูกต้อง

1.7 ข้อจำกัดของการศึกษา

1 ยากในการที่จะรู้ว่า process ลูกจะได้รับ message ที่ถูกต้องมาประมวลผล เนื่องจากไม่มี message ใดแสดงขึ้นที่เครื่องลูกเพื่อเป็นการบอกว่าได้รับ message ที่มีข้อมูลอย่างไรมาจากเครื่องแม่ ซึ่งคำสั่งที่ใช้ในการดู message ที่ถูกส่งไปยัง process ลูกใช้งานได้ยาก บางครั้งไม่สามารถแสดงข้อมูลได้โดยไม่บอกว่าเพราะเหตุใดถึงแสดง message ไม่ได้

2 มีคนศึกษาเรื่องนี้ในประเทศไทยน้อยทำให้หาที่ปรึกษาในการศึกษาการเขียนโปรแกรมแบบขนานได้ยาก ต้องติดต่อกับคนต่างชาติซึ่งในบางครั้งอาจทำให้ยากในใช้ภาษาในการติดต่อสื่อสาร

1.8 คำจำกัดความที่ใช้ในการศึกษา

PC Cluster	เป็นกลุ่มของ Computer (node)ที่มีการเชื่อมต่อเข้าด้วยกันโดยผ่าน High - Speed network หรือผ่าน LAN (Local-Area Network) โดยที่แต่ละ node อาจเป็น SMP Server หรือว่าเป็น Workstation หรือเป็น Personal Computer ก็ได้ โดยเครื่องทุกเครื่องในระบบ Cluster จะต้องมีการทำงานที่สัมพันธ์เปรียบเสมือนเป็นเครื่องเดียวกัน
Virtual Machine	ในความหมายของ library PVM คือ กลุ่มของคอมพิวเตอร์ที่มีการเชื่อมต่อกันเป็นกลุ่มคอมพิวเตอร์โดยเปรียบเสมือนเป็นคอมพิวเตอร์เครื่องเดียว
PVM	คือ library ที่เพิ่มเข้าไปใน C Compiler เพื่อให้ C Compiler สามารถทำงานแบบขนานได้ และยังใช้ในการ set กลุ่มคอมพิวเตอร์ที่เป็น virtual machine ด้วย

Master Process	เป็น processพ่อคอยจ่ายงานให้กับ และรวบรวมผลลัพธ์จาก processลูก แล้วนำมาแสดงผล
Slave Process	เป็น processลูกคอยรับงานจาก processพ่อมาประมวลผล แล้วส่งกลับมาให้ process พ่อ

บทที่ 2
หลักการที่เกี่ยวข้อง

2.1 แนวความคิดพื้นฐานของคลัสเตอร์

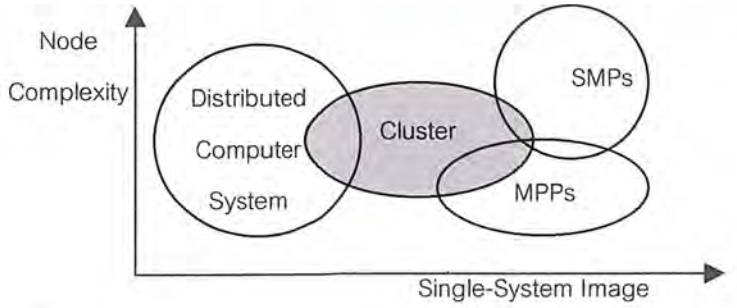
2.1.1 คุณสมบัติของคลัสเตอร์

Cluster เป็นกลุ่มของ Computer (node) ที่มีการเชื่อมต่อเข้าด้วยกันโดยผ่าน High - Speed network หรือผ่าน LAN (Local-Area Network) โดยที่แต่ละ node อาจเป็น SMP Server หรือว่าเป็น Workstation หรือเป็น Personal Computer ก็ได้ โดยเครื่องทุกเครื่องในระบบ Cluster จะต้องมีการทำงานที่สัมพันธ์เปรียบเสมือนเป็นเครื่องเดียวกัน

- Computer Nodes : เป็น Computer แต่เครื่องที่นำมาต่อกันเป็น Cluster โดยแต่ละ node จะเป็น Complete Computer คือ จะมี processor, memory, disk, I/O adapter และมี OS ประจำอยู่ที่แต่ละ node
- Single-System Image : เครื่องทุกในระบบ Cluster จะมีการทำงานที่เปรียบเสมือนเป็นเครื่องเดียวกัน โดยแต่ละเครื่องจะมีการใช้ resource ของตัวเองในการทำงาน
- Internode Connection : node แต่ละ node ในระบบ Cluster จะต้องมีการเชื่อมต่อกันด้วยระบบ network โดยอาจจะเป็น FDDI, ATM, Ethernet ก็ได้และจะต้องมี protocol ที่จะทำให้การสื่อสารระหว่างแต่ละ node มีการทำงานที่ราบเรียบไม่ติดขัด
- Enhanced Availability : Cluster เป็นทางหนึ่งที่จะเพิ่มความสามารถในการรองรับผู้ใช้งานของระบบ
- Better Performance : Cluster จะช่วยการทำงานมีประสิทธิภาพมากขึ้นและยังช่วยในการ execute โปรแกรมที่มีขนาดใหญ่ได้เร็วขึ้น

2.1.2 การเปรียบเทียบสถาปัตยกรรม

Cluster, MPP, SMP และ Distributed system เป็นสถาปัตยกรรมที่มีการทำงานที่คาบเกี่ยวกัน ดังรูป



รูปที่ 2.1 Overlapped design space of clusters,MPPs, SMPs, and distributed computer systems

ตัวอย่างที่ดีที่สุดสำหรับ Distributed System ก็คือ LAN แต่ละ node ใน LAN สามารถเป็นได้ทั้ง PC, Workstation, SMP server

SMP server เป็น computer ที่มีความซับซ้อนกว่า Cluster node เพราะว่า SMP server จะเกี่ยวข้องกับ terminal, printer, disk, external redundant array of inexpensive disks (RAID) และ tape unit ซึ่งในระบบ Cluster จะไม่รวมอุปกรณ์เหล่านี้เข้าไปอยู่ในระบบด้วย

จากรูป 2.1 ในแกน X จะเป็น degree ของความเป็นเสมือนเครื่องเดียวกันในหลายๆระดับรวมกัน ไม่ว่าจะเป็นในระดับ application, ระดับ OS kernel, ระดับ hardware ซึ่ง SMP จะมีระดับของความเป็นเสมือนเครื่องเดียวที่มากที่สุด MPP จะรองลงมาตามมาด้วย Cluster

Cluster หรือ MPP สามารถใช้ทรัพยากรคล้ายมีทรัพยากรเดียวกัน ขณะที่ distributed system จะมีการใช้ทรัพยากรของตัวเอง

ตารางที่ 2.1 การเปรียบเทียบคลัสเตอร์แบบ MPP, SMP,และ distributed sysrems

System Characteristic	MPP	SMP	Cluster	Distributed System
Number of Nodes (N)	O(100)-O(1000)	O(10) or less	O(100) or less	O(10)-O(1000)
Node complexity	Fine or Medium grain	Medium or Coarse grain	Medium grain	Wide range
Internode Communication	Message Passing or shared variables for DSM	Shared Memory	Message passing	Shared files,RPC, message passing
Job schuduling	Single run queue at host	Single run queue	Multiple queues but coordinated	Independent multiple queues
Network Protocal	Nonstandard	Nonstandard	Standard or Nonstandard	Standard
Node OS copies and Type	N(microkernel) and 1 Host OS (Monolithic)	One (monolithic)	N (homogeneous desired)	N (heterogenous)
Performanece metric	Throughput and turnaroundtime	Turnaround time	Throughput and turnaround time	Response time

ในรูปจะแสดงเปรียบเทียบของสถาปัตยกรรมต่างๆ

2.1.3 ประโยชน์และอุปสรรค์ของคลัสเตอร์

Cluster ให้ประโยชน์คือ usability, availability, scalability, และ performance/cost ratio

- Usability : แต่ละ node ใน Cluster จะมี platform ของตัวเอง user สามารถที่จะพัฒนา Application มาตามแต่ละ platform จากนั้นก็นำมาทำงานกับ Cluster ได้ Cluster จะมีการทำงานที่เพิ่ม throughput และลดเวลาในการ run ของ application
- Availability : จะเกี่ยวข้องกับเปอร์เซ็นต์ของเวลาในระบบที่ว่างสำหรับรอรับการใช้งาน อย่างเช่นใน mainframe จะให้ high availability แต่มีราคาแพง แต่สำหรับใน Cluster จะให้ high availability แต่มีราคาไม่แพง

* Processor และ memory Cluster จะมี memory หลายๆ อันและ จะมี processor หลายตัว เมื่อตัวใด fail ตัวอื่นก็สามารถที่จะทำงานแทนเพื่อให้ Cluster ทำงานต่อไปได้ ซึ่งถ้าเปรียบเทียบกับ SMP แล้ว เมื่อ processor ตัวใด fail แล้ว ระบบจะ fail ทั้งระบบ

* Disk Arrays Cluster จะมี disk หลายๆ ตัวเหตุนี้เมื่อเกิดการ fail ของ disk ตัวใดตัวหนึ่ง จะไม่ทำให้ระบบเสียไป แต่จะมีการใช้ disk ตัวอื่นแทนในลักษณะของ remote disk

* Operating System Cluster จะมี หลาย OS image อยู่ในแต่ละ node ของเครื่องใน Cluster เมื่อเกิดการ fail ของ image ใน node ใดใน cluster node อื่นๆ ก็สามารถที่จะทำงานได้ ซึ่งถ้าเปรียบเทียบกับ SMP ซึ่งจะมี OS image เดียวใน share-disk ถ้ามีการ fail ของ OS ขึ้นจะทำให้ระบบทำงานไม่ได้

- Scalable Performance : การประมวลผลโดยใช้ระบบ Cluster จะสามารถที่จะเพิ่ม node หรือ ลด node ได้ง่าย ทำให้สามารถที่จะมีได้ node เป็นร้อยๆ node ซึ่งถ้าเปรียบเทียบกับ SMP แล้วการเพิ่ม processor จะเพิ่มได้เพียงแค่จำนวนหนึ่งเท่านั้น เพราะที่การทำงานที่ซับซ้อนและที่การใช้ memory ร่วมกัน และใน SMP จะเกิดปัญหาคอขวดของข้อมูลที่เข้าออกระหว่าง memory กับ processor เพราะมีการใช้ memory ร่วมกัน

Cluster จะมี bandwidth การทำงานของข้อมูลที่เข้าออกของ memory ที่ดีกว่าเพราะมีการใช้ memory ของตัวเอง disk ของแต่ละ node สามารถที่จะรวมกันเป็น disk ที่มีขนาดใหญ่ได้ทำให้สามารถที่จะแก้ปัญหาขนาดใหญ่ ซึ่ง library ที่ใช้งานกันเช่น PVM, MPI

- Performance/Cost Ratio : Cluster ให้ประสิทธิภาพสูงในขณะที่มีราคาต่ำ ซึ่งถ้าเปรียบเทียบกับ MPP แล้ว MPP จะมีราคาสูงตั้งแต่ 10 ล้านดอลล่าร์ขึ้นไป แต่ cluster จะมีราคาถูกกว่ามาก

2.2 ระบบ PVM (Parallel Virtual Machine)

เกิดมาจากการวิจัยโปรเจคบนระบบ network จากหลายสถาบันโดยมีเป้าหมายคือ พัฒนาวิธีแก้ปัญหาให้สามารถคำนวณได้พร้อมกัน PVM ได้รวม software และ libraries ซึ่งจะเลียนแบบการคำนวณงานพร้อมกันบนคอมพิวเตอร์หลายๆตัวหลาย architecture โดยหลักของ PVM มีดังนี้

User-configured host pool: งานต่างๆที่execute บนกลุ่มคอมพิวเตอร์จะถูกเลือกโดยผู้ใช้สำหรับให้ใช้ PVM program ทั้งคอมพิวเตอร์ที่มี CPU ตัวเดียวและ multiprocessor (แชร์memory กัน)ซึ่งเป็นส่วนของ host pool ซึ่งอาจจะดัดแปลงโดยการเพิ่ม หรือ ลบ hostได้ระหว่างทำงาน

Translucent access to hardware: application program ทั้งหลายสามารถเลือกเครื่องที่ต้องการใช้ได้โดยจะให้งานกับเครื่องที่เหมาะสมมากที่สุด

Process-based computation: มีหลายงานที่อาจจะถูก execute ใน processor ตัวเดียว

Explicit message-passing model: application จะถูกแบ่งออกเป็นงานย่อยๆ โดยจะส่งและรับ message กับงานอื่นๆซึ่งขนาดของmessage จะมี limit แค่จำนวน memory ที่สามารถใช้ได้

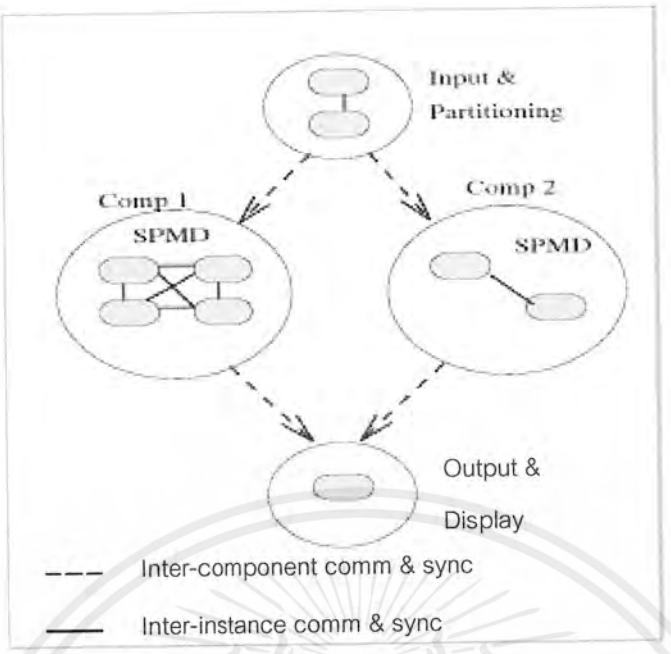
Heterogeneity support: ระบบ PVM สนับสนุนความแตกต่างกันของเครื่องต่างๆ, networks , application ที่แตกต่างกันโดยพิจารณาที่ message passing PVM อนุญาตให้ message ที่มีมากกว่า 1 datatype แลกเปลี่ยนระหว่างเครื่องที่มี data แตกต่างกัน

Multiprocessor support: PVM ใช้ message-passing ในการติดต่อกับ multiprocessors อยู่แล้ว PVM ที่มีผู้ขายนำมาขายก็ยังสามารถติดต่อกับ PVM ที่สามารถหาได้ทั่วไปได้

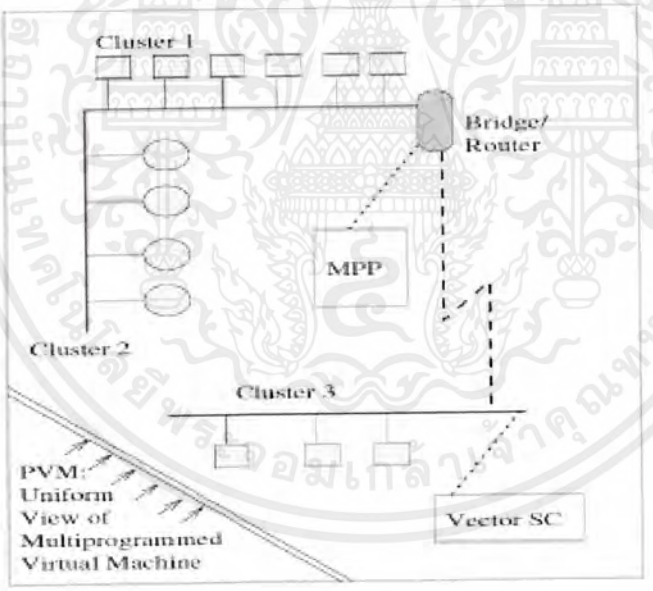
ระบบ PVM ประกอบด้วย 2 ส่วน ส่วนแรกคือ daemon ซึ่งเรียกว่า pvmd3 หรือ เรียกย่อๆ ว่า pvmd (ตัวอย่างของโปรแกรม daemon คือ mail program ซึ่ง run อยู่เบื้องหลังและจัดการการรับและส่ง electronic mail บนคอมพิวเตอร์) Pvmd3 ถูกออกแบบมาให้ผู้ใช้บางคนสามารถติดตั้ง daemon บนเครื่องได้เมื่อผู้ใช้ต้องการจะ run PVM application ขั้นแรกต้องสร้าง virtual machine PVM application สามารถทำงานจาก UNIX prompt ที่ host มีผู้ใช้หลายคนที่สามารถจัดการระบบ virtual machine ได้และผู้ใช้แต่ละคนสามารถ execute หลาย PVM application ได้ต่อเนื่องกัน

ส่วนที่สองคือ library ของ PVM มันจะประกอบด้วย ฟังก์ชั่นที่จำเป็นสำหรับการทำงานร่วมกันระหว่างงาน(tasks) ของ application ซึ่ง libraryนี้ ประกอบด้วย user-callable routine สำหรับ message passing , spawning processes, coordinating tasks และ การดัดแปลง virtual machine

การคำนวณแบบ PVM model นั้นอยู่บนพื้นฐานความคิดที่ว่า application นั้นประกอบด้วยงานย่อยๆหลายๆงาน แต่ละงานย่อยๆจะรับผิดชอบส่วนของ application บางครั้ง application จะถูกทำแบบขนานซึ่งนั้นก็คือแต่ละงานจะทำฟังก์ชั่นที่แตกต่างกัน กระบวนการนี้จะถูกเรียกว่า functional parallelism method ที่เหมือนกันของการทำงานแบบขนานถูกเรียกว่า data parallelism ซึ่งแต่ละงานเพียงรู้และแก้ปัญหาส่วนเล็กของ data นี้ถูกอ้างถึงเหมือนการทำงานแบบ SPMD (single-program multiple-data)model PVM สนับสนุนทั้งสองวิธีหรือแบบรวมกันของทั้งสองวิธีโดยขึ้นอยู่กับฟังก์ชันทั้งหลาย เช่น งานทั้งหลายอาจจะ execute แบบ parallel และอาจจะจำเป็นต้องรอ สัญญาณนาฬิกา หรือ การแลกเปลี่ยน data กัน



รูปที่ 2.2 PVM computing model



รูปที่ 2.3 PVM Architecture Overview

ปัจจุบัน PVM สนับสนุนภาษา C ,C++ ,และ Fortran แต่ส่วนใหญ่จะใช้กับระบบภาษา C ภาษา Fortran ถูกนำมาใช้เป็น subroutines มากกว่าที่จะเป็นฟังก์ชัน

ทุกงานใน PVM ถูกกำหนดโดยตัวเลข task identifier (TID) ตัวเลขนี้ต้องไม่ซ้ำกันซึ่งจะถูกกำหนดโดย pvmd ผู้ใช้ไม่ต้องกำหนดเอง PVM มีหลาย routine ที่จะให้ค่า TID ดังนั้นผู้ใช้สามารถระบุงานอื่นๆในระบบได้ มีหลาย application ที่ซึ่งคือ กลุ่มของงาน(group of tasks) มีหลายกรณีที่ผู้ใช้เหมือน

ระบุงานโดยเบอร์ 0-(p-1) ซึ่ง p คือจำนวนของงาน เมื่องานหลายๆงานรวมเป็นกลุ่ม มันจะถูกกำหนดเลขเฉพาะแทนงานภายในกลุ่มตัวเลขนี้จะเริ่มที่ 0 ขึ้นไป

ตัวอย่างทั่วไปสำหรับการ programming กับPVM คือ ผู้ใช้เขียน หนึ่งหรือมากกว่าsequential program ด้วยภาษา C,C++,Fortran77 นั้นจะใส่ PVM library ติดไปด้วย โปรแกรมที่มีลักษณะเช่นเดียวกันจะถูกสร้างเป็นงาน(task) โปรแกรมเหล่านั้นจะถูกคอมไพล์แต่ละ architecture ใน host pool และผลจะถูกเก็บไว้ที่ตำแหน่งเครื่องใน host pool สามารถเข้าได้ การexecute สามารถทำได้จากเครื่องภายใน host pool กระบวนการต่อมาก็จะเริ่มทำงานอื่น จนในที่สุดผลก็จะถูกรวบรวมคำนวณและแลกเปลี่ยนข้อมูลกับส่วนอื่นๆ

บทที่ 3

หลักการทำงานของ PVM

3.1 ส่วนประกอบ

3.1.1 ตัวบ่งบอกงาน (Task identifiers, TID)

PVM ใช้ตัวบ่งบอกงาน (TID) ในการบอกชื่อของงาน (task) และกลุ่มของงาน (task) ที่อยู่ในเครื่องเสมือน(virtual machine) ตัวบ่งบอกงาน(TID) จะประกอบด้วย 4 ส่วนดังรูป 3.1 ซึ่งถ้ามีการทำงานจำนวนมากตัวบ่งบอกงาน(TID) จะถูกทำให้เป็น integer 32 บิต ซึ่งสามารถขยายให้ใหญ่กว่าเดิมได้

field S,G และ H จะมีความหมายโดยรวม ในแต่ละ pvmd จะแปลความหมายได้แบบเดียวกัน โดย S จะบอกชื่อ(address) ของ pvmd, L ใช้สำหรับลบตัวบ่งบอกงาน(TID) ในส่วนของ H ประกอบด้วยหมายเลขโฮสต์ซึ่งจะสอดคล้องกับเครื่องเสมือน(virtual machine) โดยค่ามากที่สุดที่จะใช้เป็นหมายเลขโฮสต์ได้ คือ $2^{H}-1$ (4905) โดยแปลงจากหมายเลขโฮสต์ไปเป็นโฮสต์แต่ละ pvmd โดยจะดูจากข้อมูลที่มีอยู่ในตารางโฮสต์(Host Table) และ G ใช้อ้างอิงถึงกลุ่ม task

รูปที่ 3.1 Generic task id

ในการออกแบบตัวบ่งบอกงาน(TID) สามารถเพิ่มเติมเพื่อให้ดี โดย TID สามารถถูกกำหนดขึ้นโดย pvmd ประจำเครื่องได้เลยโดยไม่ต้องมีการติดต่อระหว่างโฮสต์กันเลย ช่องว่างที่ถูกจองไว้จะใช้สำหรับรหัสของข้อผิดพลาด(error code)

ตารางที่ 3.1 ตัวบ่งบอกงาน

Use	S	G	H	L
Task identifier	0	0	1..H	1..L
Pvmd identifier	1	0	1..H	0
Local pvmd (from task)	1	0	0	0
Pvmd' from master pvmd	1	0	0	0
Multicast address	0	1	1..H	0..L
Error code	1	1	(small neg. number)	

3.1.2 สถาปัตยกรรมของ PVM

PVM เป็นสถาปัตยกรรมที่มีการทำงานระหว่างเครื่องคอมพิวเตอร์ที่มีการทำงานที่ต่างกันไม่ว่าจะเป็นความแตกต่างกันของตัวเครื่องคอมพิวเตอร์เองหรือมีระบบปฏิบัติการแตกต่างกัน และยังมีการทำงานกับเครื่องที่มีการประมวลผลที่ไม่ตรงกันกับเครื่องในระบบ PVM เอง

PVM จะช่วยในการแก้ปัญหาเกี่ยวการแปลงลักษณะของข้อมูลให้เครื่องในระบบ PVM ใช้งานกันได้ทำเครื่อง

3.1.3 รูปแบบของข้อความ(Message Model)

PVM daemons (pvmd) และงาน(task) สามารถรวมกันและส่งข้อมูล(message)ได้อย่างอิสระ ข้อมูลสามารถแปลงไปโดยใช้ XDR เมื่อเกิดการส่งข้อมูลระหว่างโฮลต์ที่รูปแบบข้อมูลไม่เข้ากัน (incompatible)

ผู้ส่งข้อมูลจะไม่ทราบว่าได้รับข้อมูลจากผู้รับ ข้อมูล(meesage) ที่ถูกส่งมาจะถูกเก็บไว้ใน buffer ซึ่งปลอดภัยต่อการลบหรือลักลอบเอาไปใช้ PVM จะรับประกันการส่งข้อมูลที่แน่นอนโดยจะกำหนดจุดปลายทางทำให้ข้อมูลที่ส่งถูกรักษาตลอดทั้งระบบ

3.1.4 การแจ้งเหตุที่เกิดขึ้นแบบไม่ต่อเนื่อง(Asynchronous Notification)

PVM จะมีการจัดแบ่งแจ้งเหตุการณ์ที่เกิดขึ้น(notification) เพื่อใช้ในการรองรับความผิดพลาด(fault tolerance) งานหรือโปรแกรมต่างๆจะสามารถร้องขอต่อระบบในการส่งข้อมูลเกี่ยวกับเหตุการณ์ที่เกิดขึ้นได้แบ่งเป็น 3 เหตุการณ์

ตาราง 3.2 แสดงเหตุการณ์ที่เกิด

Type	Meaning
PvmTaskExit	Task exits or crashes
PvmHostDelete	Host is deleted or crashes
PvmHostAdd	New hosts are added to the PVM

โดยคำร้องดังกล่าวจะเก็บไว้ใน pvmd เพื่อใช้ในการเฝ้าดู กรณีร้องขอในรีโมตอีเวนท์(เกิดขึ้นที่ต่าง host กัน) มันจะถูกรักษาโดยโฮสต์ทั้งสอง pvmd ของรีโมตโฮสต์จะส่งข้อมูล ถ้ามีเหตุการณ์เกิดขณะที่ pvmd ของเครื่องปัจจุบันจะส่งข้อมูลบอกถ้ารีโมตโฮลต์หยุดทำงาน

3.1.5 PVM Daemon and Programming Library

PVM Daemon เป็น pvmd ชนิดหนึ่งที่ทำงานในแต่ละโฮลต์ของเครื่องเสมือน(virtual machine)โดยให้บริการในการกำหนดเส้นทางและควบคุมโดยมันจะควบคุมและตรวจสอบความผิดพลาด pvmd ที่ว่างจะคอยตรวจสอบการทำงานของเครื่องเสมือนตลอดการทำงาน ถ้าเกิดการหยุดทำงานของโปรแกรม pvmd จะทำงานต่อเพื่อแก้ไขและตรวจสอบข้อผิดพลาดของโปรแกรม

pvmd แรกคือ พ่อ(master) และมี pvmd ตัวอื่นที่ทำงานต่อจากพ่อ(master) เรียกว่า

ลูก (slaves) ซึ่งจะทำงานร่วมกันและมีเพียง master เท่านั้นที่สามารถแก้ไขเพิ่มเติมและกำจัดโฮสต์ออกจากเครื่องเสมือนได้

ไลบรารี่ libpvm จะอนุญาตให้งานต่างๆ มีการติดต่อกับ pvmd ซึ่งจะประกอบด้วยฟังก์ชันของเตรียมข้อมูลก่อนส่งและขยายข้อมูลที่ได้รับและฟังก์ชันของ PVM syscall โดยใช้การร้องขอข้อมูลจาก pvmd

โดยในระดับสูงสุดของ libpvm จะรวบรวมโปรแกรมการเชื่อมต่อฟังก์ชันที่ถูกเขียนขึ้นเท่าที่จำเป็น และระดับล่างจะจัดเก็บและแยกซึ่งสามารถปรับปรุงหรือแทนที่เครื่องคอมพิวเตอร์ที่อยู่ในสภาพแวดล้อมใหม่

3.2 ข้อความ(Messages)

3.2.1 Fragments and Buffers

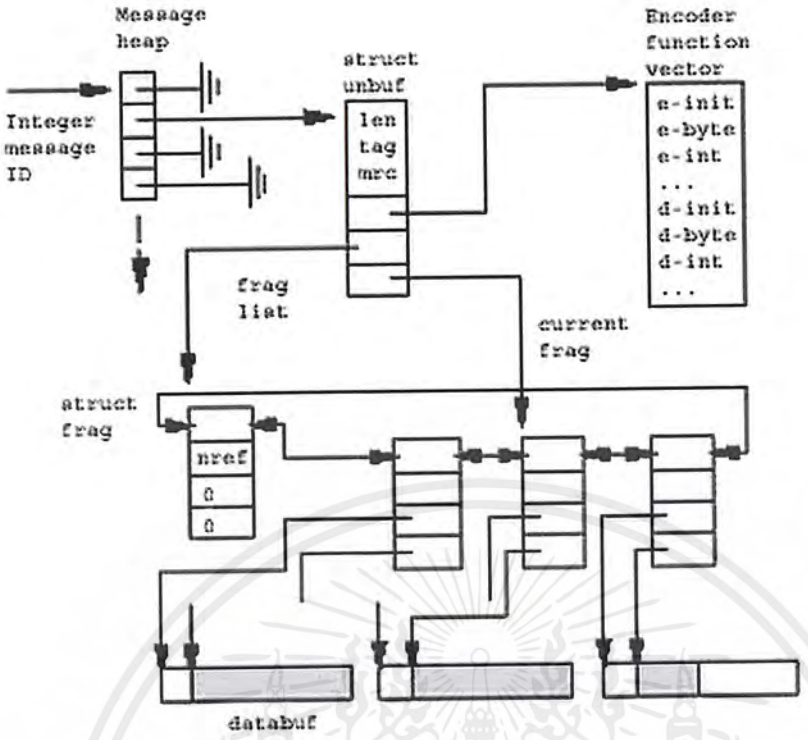
Pvmd และ libpvm จะทำการสร้างบัฟเฟอร์ซึ่งจะเก็บข้อมูลขนาดใหญ่ของข้อมูลที่ปรับเปลี่ยนได้(dynamic data) และทำการแบ่งการใช้ข้อมูลอย่างมีประสิทธิภาพเพื่อที่จะส่งข้อมูลไปควรจะหลีกเลี่ยงการทำสำเนา(copy) ข้อมูลที่อยู่ในบัฟเฟอร์

ข้อมูลที่ถูกส่งใน PVM จะไม่มีการประกาศ ค่าสูงสุดของความยาวไว้ก่อนโดยจะมีฟังก์ชั่นที่ทำการจัดสรรหน่วยความจำและใช้บัฟเฟอร์ในการเก็บข้อมูล

frag descriptor เก็บ pointer (fr_dat) ของข้อมูลไว้ใน block และ length (fr_len) มันจะเก็บ pointer (fr_buf) ที่ชี้ไปที่ข้อมูลในบัฟเฟอร์และความยาวสูงสุด (fr_max) ซึ่งจะเก็บส่วนที่ว่างไว้รักษาข้อมูล

3.2.2 ข้อมูลใน Libpvm

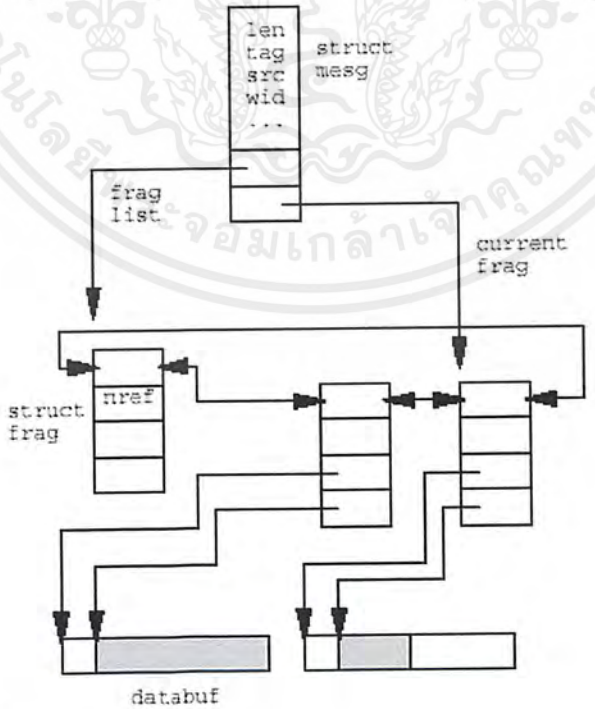
Libpvm จะจัดหาฟังก์ชันในการเตรียมข้อมูลก่อนส่ง(pack message) ซึ่งมี 5 ขั้นตอนสำหรับเข้ารหัส(encoder) และ ถอดรหัส(decoder) ซึ่งข้อความในบัฟเฟอร์จะถูกจัดการโดย libpvm เมื่อมีข้อมูลใหม่ตัวเข้ารหัส(encoder)จะเข้ารหัสโดยพิจารณาจากรูปแบบของข้อมูลที่เข้ามา เมื่อผู้รับได้รับจะทำการถอดรหัสโดยพิจารณาจาก encoding field ดังรูป



รูปที่ 3.2 แสดงข้อมูลที่ถูกเก็บอยู่ใน libpvm

3.2.3 ข้อมูลที่อยู่ใน pvmd

ข้อมูลที่อยู่ใน pvmd ถูกใช้ด้วย struct mesg



รูปที่ 3.3 Message storage in pvmd

ซึ่งจะencode สำหรับ sign และ unsign integer และ sting ที่ใช้ใน libpvm foo format โดย integer จะมี 4 บิต เท่านั้น

3.2.4 Pvmd Entry Points

ข้อมูลของ pvmd บรรจุในแพ็คเกตใน loclinpkt() ถ้ามาจากงานของเครื่องตัวเอง ,และจะอยู่ใน netinpkt() ถ้ามาจาก pvmd อื่นหรืองานในเครื่องอื่นซึ่งจะอยู่ 1 ใน 3 ดังนี้

ตารางที่ 3.3 แสดง message ที่บรรจุในแพ็คเกต

Function	Messageform
loclentry()	Local tasks
Netentry()	Remote pvmds
Schentry()	Local or remote special tasks (Resource manager, Hoster, Tasker)

ถ้า message ไม่อยู่ในนี้ก็จะตัด message นั้นทิ้งไป

3.3 PVM Daemon

3.3.1 การเริ่มทำงาน

Pvmd จะกำหนดตัวมันเองให้เป็นพ่อ(master) หรือลูก(slave) และจะสร้าง socket ที่ใช้ในการเชื่อมต่อกับ pvmd อื่นๆ โดย pvmd ของพ่อจะอ่านไฟล์ที่มีรายชื่อของโฮสต์ ส่วนของ pvmd ของลูกจะสร้างการเชื่อมต่อกับ mater pvmd หลังจากเชื่อมต่อเรียบร้อยแล้ว pvmd จะเข้าระบบ loop ของฟังก์ชัน work() เพื่อรอรับคำสั่งทำงาน

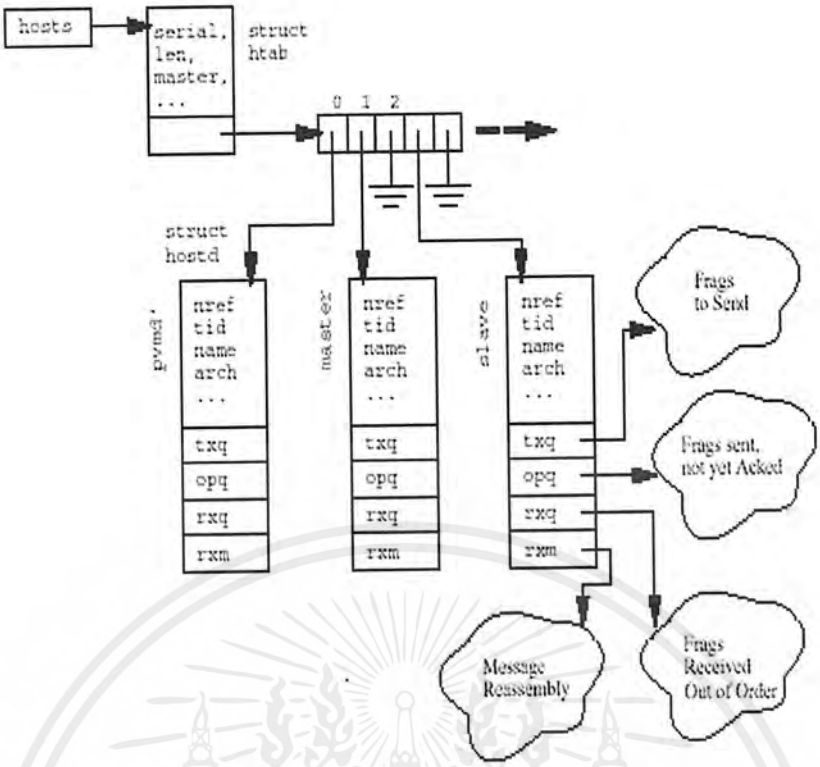
3.3.2 การปิด Daemon

Pvmd ถูกปิดเมื่อมันถูกลบทิ้งจากเครื่องเสมือน(virtual machine) หรือมีการขาดการติดต่อกับ master pvmd แล้วจะมีการทำงานอยู่ 2 อย่าง

1. กำจัดทุก task ที่ ทำงานอยู่ด้วย SIGTERM
2. ส่ง final shutdown message ไปในทุก pvmd ที่อยู่ในตารางโฮสต์(host table)

3.3.3 ตารางโฮสต์(Host Table) และกรปรับแต่งะบบ(Machine Configuration)

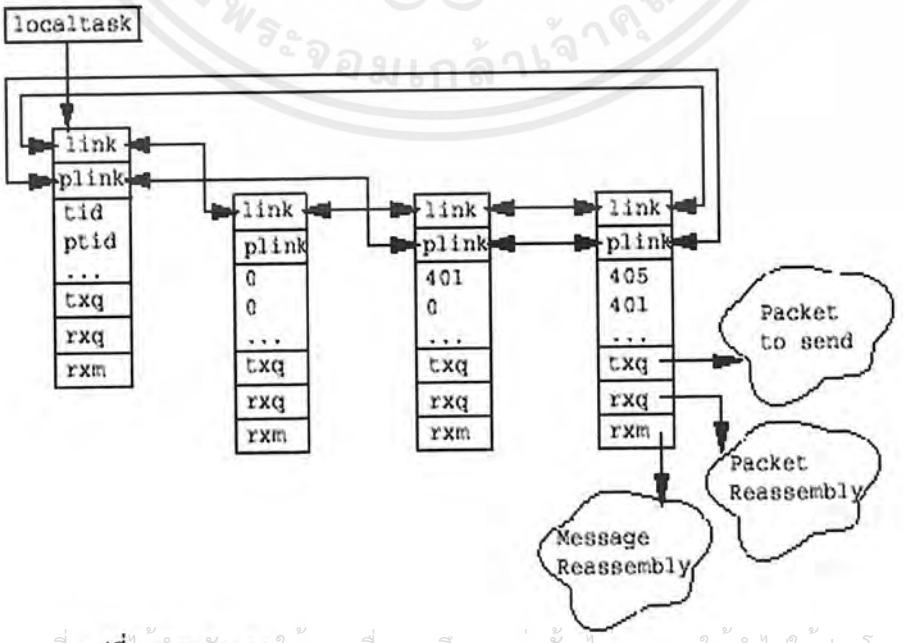
ตารางโฮสต์จะอธิบายโครงสร้างของเครื่องเสมือน(Virtual machine) ซึ่งจะมี ชื่อ ที่อยู่ และสถานะที่ทำการเชื่อมโยงข้อมูลสำหรับแต่ละโฮสต์



รูปที่ 3.4 ตารางโฮสต์สร้างจาก struct htab และ struct host

ตารางโฮสต์จะถูกสร้างโดย pvmd พ่อ(master pvmd) และจะมีส่งสัญญาณสื่อสารกันระหว่าง pvmd ในเครื่องเสมือน(virtual machine) สำหรับการตั้งค่าของเครื่องเสมือน(virtual machine) เมื่อเกิดการเปลี่ยนแปลงขึ้นจะต้องมีการส่งสัญญาณถึงกันเพื่อใช้ในการบอกข่าวสารระหว่าง pvmd กัน โฮสต์ไฟล์เป็นไฟล์ที่ใช้ในการเริ่มต้นการทำงานของ pvmd พ่อ(master pvmd) ใช้ในการเพิ่มโฮสต์ลงไปในเครื่องเสมือน

3.3.4 งาน(tasks)

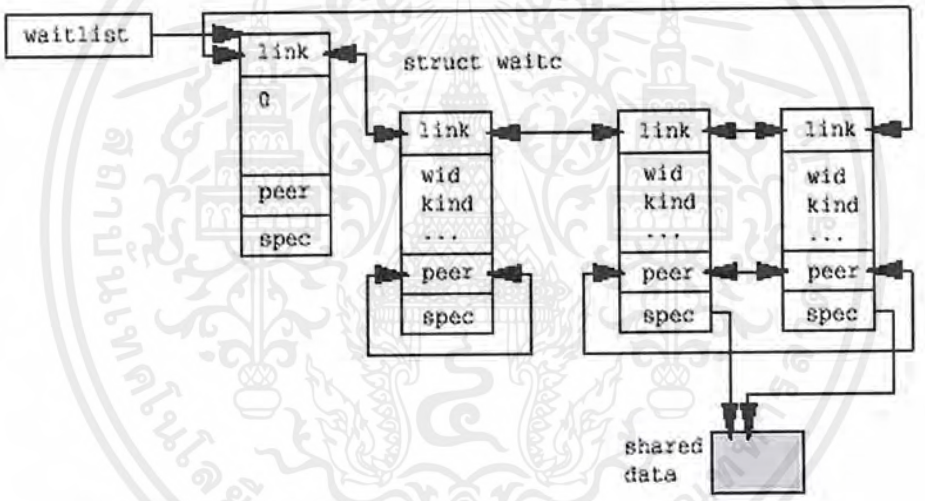


รูปที่ 3.5 Task table

แต่ละ pvmd จะประกอบด้วยรายการของงานทุกๆ งานและจะเรียงตามหมายเลขงาน (id task) ส่วนมากจะอยู่ในลิสต์ที่สองที่เรียงตามหมายเลขโพรเซส(process id) โดยจะอยู่ในส่วนเดียวกันคือ ส่วนงานประจำเครื่อง(local task) PVM จะทำการตรวจสอบข้อมูล (debugging)โดยสร้างงาน(task) ชนิดพิเศษที่เรียกว่า "Tasker" tasker เริ่มทำงานในส่วนงานอื่นๆ (exces) โดยทั่วๆไป debugger คือกระบวนการที่ควบคุมการประมวลผลของกระบวนการอื่น สามารถอ่านเขียน หยุดคำสั่งได้

3.3.5 Wait Contexts(waitc)

Pvmd ใช้ wait context เมื่อมีการขัดจังหวะ(interupted) ซึ่ง PVM จะไม่ส่งข้อมูลไปทุกๆ เครื่อง(muticast) แต่จะแสดงว่ามีการขัดจังหวะออกมาพร้อมๆกันทุกเครื่อง เมื่อ pvmd รับ syscall จาก task และมี pvmd อื่นมาชน ก็จะมีการรอเกิดขึ้นและจะมีการ save ไว้ใน pvmd แล้วกลับไปทำงานใน work() เมื่อคำตอบมาถึง pvmd จะ stashed ใน waitc syscall และ reply ของ task โดย waitc คือ ลำดับเลขที่จะส่งไปยัง header ด้วยการร้องขอหรือ reply



รูปที่ 3.6 Wait context list

เมื่อโฮสต์ fail หรือ task exit ,pvmd หา wait list สำหรับหยุดใน TID และออกจากการทำงาน โดย waitc จาก host ที่ตายหรือ task ที่ถูก block จะไม่ถูกลบทิ้งไป ถ้า wa_tid เท่ากับศูนย์มันจะเข้าใกล้ระบบป้องกันแบบ recycle

3.3.6 การค้นหาข้อผิดพลาดและการกู้คืน(Fault Detection And Recovery)

Fault Detect เริ่มจาก pvmd-pvm protocal เมื่อ PVM พบความผิดพลาดจะเรียก hostfailenter() ที่ scan จาก waitlist และกำจัดการทำงานที่รอทั้งหมดใน downhost โดย pvmd ครอบคลุมถึง pvmd นอก master เมื่อ slave losed มันจะปิดตัวเองซึ่งจะมี algolithm ยืนยันอยู่แล้ว เพราะmasterจะไม่มีการชน เพราะมันไม่มีเส้นทางที่ master ใช้ร่วมกับ pvmd อื่นๆ คือมันจะทำการปิด pvmd ที่ซ้อนทับกัน

Pvmd' → slave: (exec) $PVM_ROOT/lib/pvmd –s –d8 –nhonk 1 80a9ca95:0f5a
4096 3 80a95c43:0000

slave → pvmd': ddpro<2312> arch<ALPHA> ip<80a95c43:0b3f> mtu<4096>

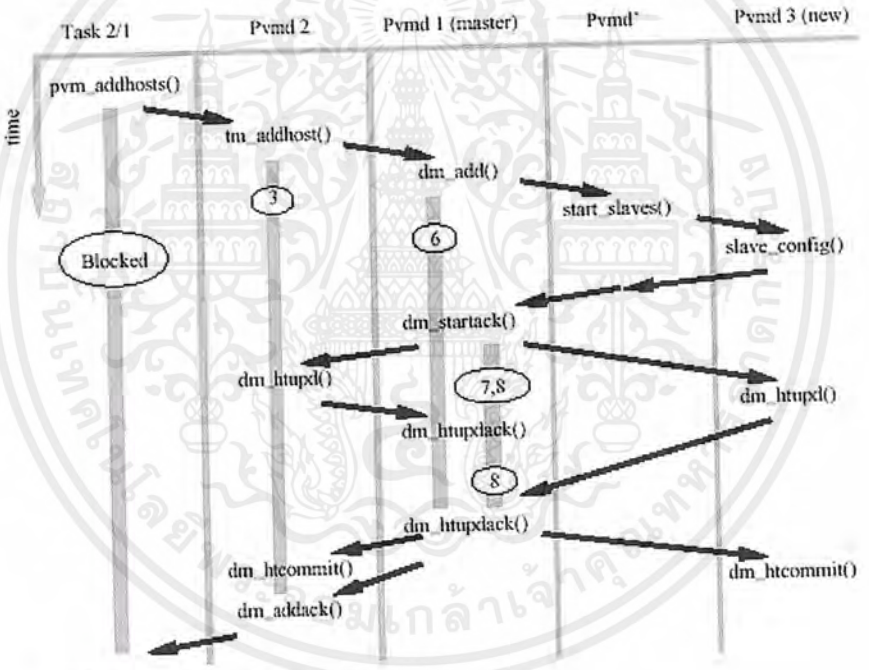
pvmd' → slave: EOF

3.3.7 Pvmd

เป็นเงาของ pvmd ที่ทำงานบน master host ใช้สำหรับ master ในการสร้าง slave pvmd

3.3.8 การเริ่มต้นการทำงานของ pvmd ลูก(slave pvmd)

Slave pvmd เริ่มเมื่อ task ยุ่งเหยิงโดยมีจุดมุ่งหมายในการที่จะทำงานบน host ใหม่ด้วยลักษณะที่คล้ายกันสามารถเพิ่ม ลด เปลี่ยนแปลงได้ โดยทั่วๆไป machine ควรจะมีค่าความปลอดภัยและมีการทำงานติดตั้งที่เร็วและไม่ต้องมีการใช้ password



รูปที่ 3.7 Timeline of addhost operation

จากรูปแสดง host ทำงานกับ machine task เรียก pvm_addhost()แล้วส่งไปยัง pvmd ซึ่งมันจะสร้าง DM_ADD ไปยัง master master pvmd จะสร้าง host table ใหม่สำหรับแต่ละการร้องขอจาก host โดยดู IP address และ set option ของ host ที่เข้าออก host descriptor จะเก็บ waitc_add structure แต่ยังไม่ใส่ไว้ใน host table และ master จะแยก pvmd ให้ไปทำงานที่เลวกว่าแล้วจึงส่งมันไปใน list ของ host ซึ่ง pvmd จะใช้ rsh,rexec() หรือในการเริ่มแต่ละ pvmd ส่ง parameter ซึ่งการเปลี่ยนแปลง slave และ pvmd' มีรายละเอียดดังนี้

Address ของ master กับ slave pvmd จะส่งไปยังcommand line และเมื่อมีการเปลี่ยนแปลงจะรอ EOF จาก pvmd' และจะติดต่อไม่ได้แต่จะทำงานในสถานะ

(runstate=PVMSTARTUP) หลังจากรู้ status จากการเปลี่ยนแปลงล่าสุด จาก master pvmd ถ้าไม่มีการ configered ภายใน 5 นาที(PARAMETER DDB AILTIME)จะสรุปว่ามีปัญหากับ master pvmd และจะเลิกการทำงาน เมื่อ host มีการทำ parallel pvmd จะส่งข้อมูลหรือ error ใน DM_STARTACK ไปที่ master pvmd ถ้าสมบูรณ์แล้ว descriptor จะใช้ wait context เมื่อมีการส่ง task เข้ามาใน hoster ที่เป็นลูกของ master pvmd เมื่อมันได้รับ DM_ADDแล้ว pvmd ใช้งานไม่ได้ SM_STHOSTmessageจะทำงานแทนโดยจะถูกส่งไปที่ hoster และ SM_STHOSTACK message ไปยัง master pvmd ทฤษฎีดังกล่าวใช้ slave pvmd แทนได้แต่ hoster จะไม่เข้าใช้ โปรโตคอล และถ้า hoster task fails ขณะปฏิบัติงาน pvmd จะเรียกใช้ wait context ซึ่งคือ เริ่มจาก slave และจะส่ง DM_ADDACK message ในการแสดงระบบความผิดพลาด หลังจาก slaveเริ่มทำงาน masterจะส่ง DM_SLCONF message ไปทั้ง parameter ที่ไม่รวม โปรโตคอล ตอนเริ่มต้น และจะกระจาย DM_HTUDD message ไปยังทุกความถี่ที่เข้ามาใหม่ใน slave เมื่อได้รับข้อความแต่ละ slave จะรู้และจะสร้าง virtual machine ใหม่ขึ้นมา master จะรอ DM_HTUPDACK จากทุก slave โดยกระจาย HT_COMMIT message แล้วไปใส่ไว้ใน host table อันใหม่ ในที่สุด master จะส่ง DM_ADDACK ซึ่งจะตอบคำร้องขอ และจะให้ ID.host

3.3.9 Resource Message (RM)

เป็น PVM task ที่มีหน้าที่สร้าง task และตาราง host ที่บอกหน้าที่ ตารางทั่วไปจะมีเงื่อนไขมากมายเพื่อให้ผู้ใช้เข้าใจส่วนประกอบของโปรแกรมและให้มีประสิทธิภาพมากที่สุด number of RMs สามารถเปลี่ยนแปลงจาก หนึ่งต่อ virtual machine ไปเป็น หนึ่งต่อ pvmd โดย RM จะทำงานบน master host เมื่อ master pvmd ทำงาน ซึ่ง slave pvmd จะไม่มี RM ของมันเอง โดย task จะเชื่อมต่ออย่างลับๆกับ virtual machine ที่ไม่มีค่า RM ของ pvmd ที่ทำการเชื่อมต่อ โดย task จะ spawn จากภายในระบบที่มีการถ่ายทอด RM ของ task พ่อ แต่ task ถ้า มี RM จะร้องขอไปยัง pvmd และแสดงเส้นทางใน RM แทน โดยมี libpvm function เป็นตัวกลาง

ตารางที่ 3.4 แสดงการใช้ฟังก์ชัน addhost, delhost, spawn

Libpvm function	Default Message	RM Message
pvm_addhost()	TM_ADDHOST	SM_ADDHOST
pvm_delhost()	TM_DELHOST	SM_DELHOST
pvm_spawn()	TM_SPAWN	SM_SPAWN

3.4 Libpvm Library

3.4.1 ภาษาที่สนับสนุน

Libpvm เขียนโดย ภาษา C สามารถรองรับ C , C++ ซึ่ง Fortran Library , libpvm3.a เป็นกลุ่มของ wropper function

3.4.2 การติดต่อกับ Pvmd

ขั้นแรกเรียก libpvm function , pvm_beatask() ซึ่งเป็น Libray เริ่มต้น และเชื่อมต่อ task กับ pvmd การเชื่อมต่อ task ที่เป็นความลับ จะแตกต่างกันเล็กน้อยกับแบบแรก(คือ spawned task) pvmd จะมีการประกาศ address ของ socket ซึ่งมีรูปแบบ / tmp/pvmd.uid , uid คือ เลขผู้ใช้ภายใต้ pvmd ที่ใช้ ซึ่งจะมีรูปแบบของมัน 7f000001:06f7or /tmp/aaa014138 มันก็คือ IP address และหมายเลข Port ของ socket หรือเป็น path ใน Unix-domain socket spawned task และเมื่อ task spawn โดย pvmd descriptor จะสร้าง pvmsock ตลอดการทำงานซึ่งมันสามารถจะ stash ข้อความทั้งหมดก่อนที่จะติดต่อไม่ได้ ตลอดการ reconnection task จะแสดงว่าตัวมันเองกับ pvmd ยังทำงานอยู่ด้วย PID จึงจะมีการอนุญาตให้ติดต่อภายในกระบวนการร้องขอการเชื่อมต่อใหม่ ซึ่ง pvmd จะผ่านค่า PID-> PVMEPID แล้ว PVM จึงจะสามารถควบคุม task ได้ Pvm_beatask() สร้าง TCP socket และเชื่อมต่อกับ pvmd ซึ่งแต่ละอันจะป้องกันผู้ใช้อื่นที่แปลกปลอมเข้ามา ซึ่งสามารถทำโดย /tmp ตามด้วยชื่อเจ้าของงาน แล้วนำไปเก็บไว้ file สำหรับ protocol serial (TDPROTOCOL) จะนำมาเปรียบเทียบเมื่อ task เชื่อมต่อกับ pvmd หรือ task อื่นๆ โดยตัวเลขจะมีค่าเพิ่มขึ้นเมื่อเปลี่ยน โปรโตคอล เพราะมันจะไม่ซ้ำกับ โปรโตคอล อันก่อน

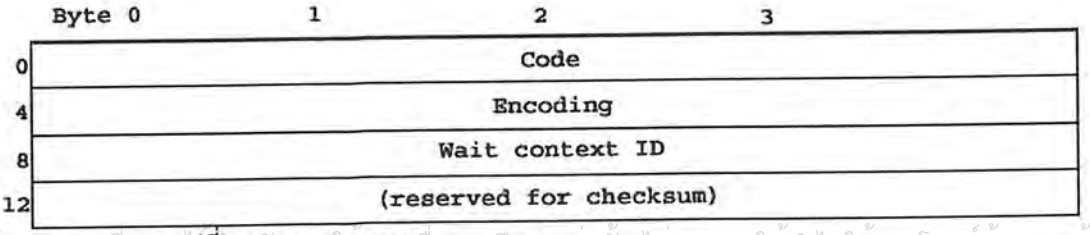
3.5 โปรโตคอล

PVM สื่อสารบน TCP ,UDPและ Unix-domain socket ซึ่ง support PVM ที่ส่งข้อความอย่างมีประสิทธิภาพและยังสนับสนุน multicasting และคำนึงถึง priority ของข้อมูล ซึ่งไม่จำเป็นต้องติดตั้งก่อนการใช้ซึ่ง PVM โปรโตคอล จะพิจารณา 3 อย่างการเชื่อมต่อ

- ระหว่าง pvmd
- ระหว่าง pvmd และ task
- ระหว่าง task

3.5.1 Message

Pvmd และ libpvmใช้ message header เหมือนกันโดย code จะประกอบด้วยจำนวนเต็ม และ libpvm มีการ Encoding ในการผ่านรหัสโดยเก็บไว้ในรูปแบบที่แตกต่างกัน



รูปที่ 3.8 Message header

Pvmd set Encoding เป็น 1 แล้วใช้ wait context ผ่านไปยัง wait id's มีหลายค่าร่วมกับข้อมูล บางที task .ใช้ wait id's โดย checksum ทำหน้าที่รักษาข้อมูล

3.5.2 Pvmd-Pvmd

Pvmd daemons ติดต่อผ่านทาง UDP socket UDP เป็นการส่งข้อมูลที่อาจสูญหาย การพัฒนา TCP ใน 3 ปัจจัย ที่ทำให้เสื่อมประสิทธิภาพ

- First-Scalability

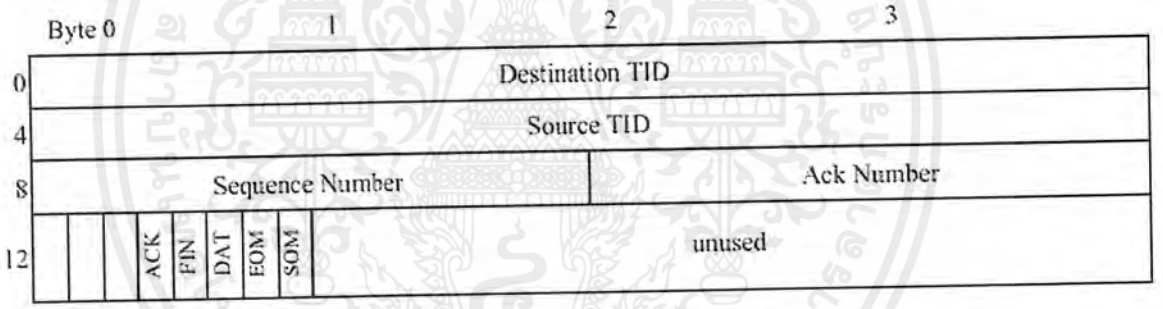
ใน virtual machine ของ N host แต่ละ pvmd ต้องติดต่อกับ pvmd อื่น N-1 ซึ่งแต่ละอันโดยใช้ TCP เชื่อมต่อกันทำให้เปลืองเนื้อที่ (file) บางการปฏิบัติการมีการกำหนดการเปิด file เช่น 32 บิต ส่วน UDP socket สามารถสื่อสารด้วยด้วยเลขของ UDP socket

- Second overhead

N pvmd ต้องการ N(N-1)/2 TCP ในการเชื่อมต่อ ซึ่งการเชื่อมแบบนี้ทำให้สิ้นเปลืองทรัพยากร

- Third-fault tolerance

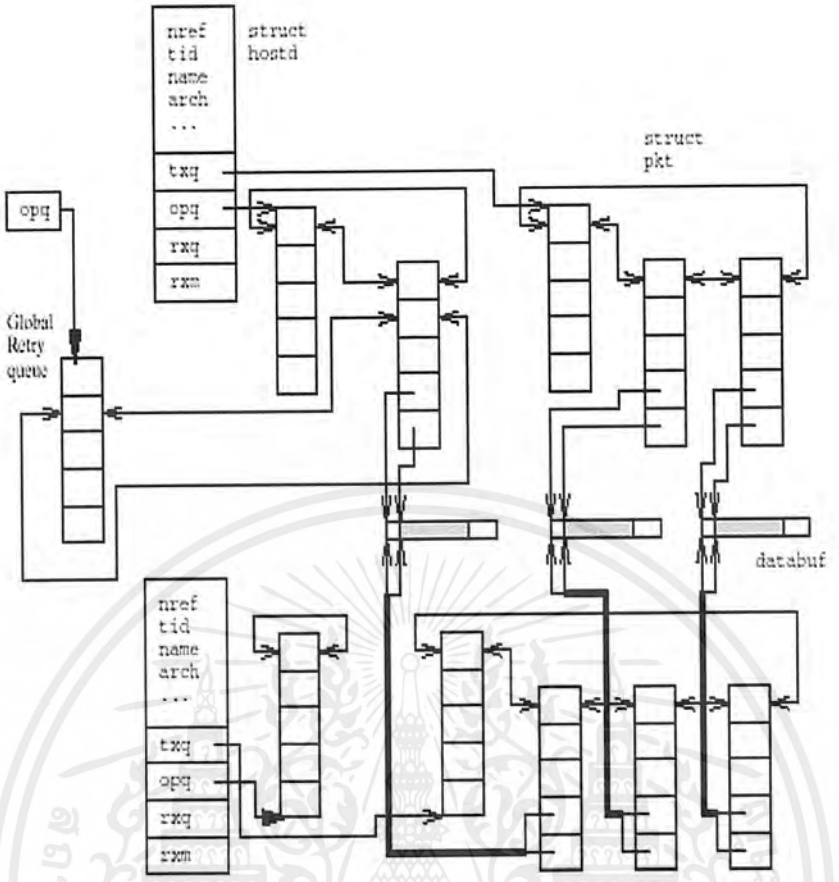
การสื่อสารจะมีการตรวจสอบเมื่อ pvmd อื่นมีการชนของข้อมูลที่ส่งหรือระบบเครือข่ายไม่ทำงาน



รูปที่ 3.9 Pvmd-Pvmd packet header

ข้อมูลจะถูกส่งในตามลำดับความสำคัญในแต่ละไบต์ จะมีการบรรจุจุดหมายปลายทางไว้ใน TIDs ของ packet และจะมีการส่งการตอบรับ(Acknowledgement) เริ่มจาก 1 เพิ่มไปถึง 65535 จึงไม่เป็น 0 SOM(EOM) ทำก่อนหรือหลัง fragment ของข้อมูล ซึ่งภายในจะมี SOM (EOM)จะถูกใช้ใน task และ pvmd

สำหรับกำหนดขอบเขต DAT ถ้ากำหนดข้อมูลถูกบรรจุอยู่ใน packet และ sequence number มีค่า ถูกต้อง จะติดตั้ง Acknowledgement number ถูกบันทึกทำให้เชื่อมโยงกับ DAT ใน packet FIN ใช้เมื่อ pvmd สิ้นสุดการทำงานโดย ACK และ Acknowledgement number จะทำการติดต่อกับ FIN packet ผ่านทาง sequence number สุดท้าย packet จะส่งทั้ง FIN,ACK และถ้ามีการเชื่อมต่อกับ pvmd อื่นก็จะบันทึกไว้ใน host tableจะมีโปรโตคอล ที่ใช้ตาม field ของ struct hosted

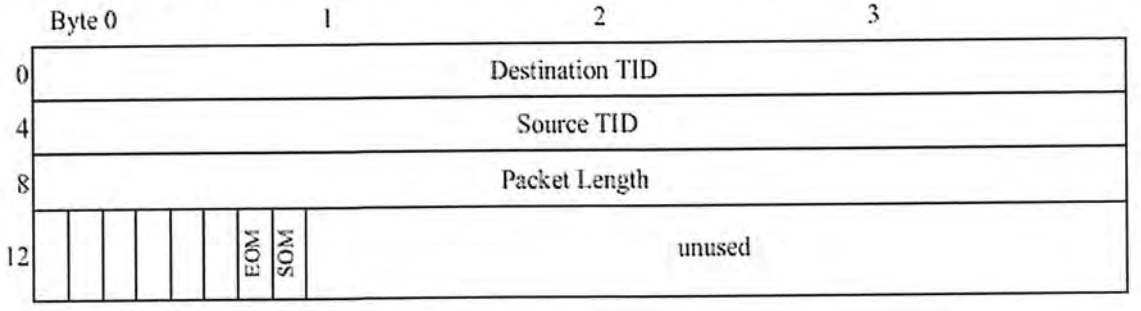


รูป3.10 Host descriptor with queues

แสดง host ส่งและ packet สำคัญใน queues packet จะรอคอยการส่ง FIFO hd_txg ของ queues ด้วย routing code ถ้าไม่ได้รับ routing code จะทำการส่งผ่านไปยัง queues อื่นๆทันที โดยโปรโตคอล มีความสำคัญมากเป็นทวีคูณในการพัฒนาประสิทธิภาพของ network

3.5.3 Pvmd_task and Task_Task

Task ติดต่อกับ pvmd และกับแต่ละ task ผ่านทาง TCP socket เลือกใช้ TCP เพราะมันมีการส่งที่เสียหายยากซึ่ง UDP อาจสูญเสีย packet การสื่อสารที่ไม่แน่นอนต้องการทั้ง TCP และ UDP แต่ task ไม่สามารถขัดจังหวะขณะมีการทำงานของ I/O เราจึงไม่ใช้ UDP



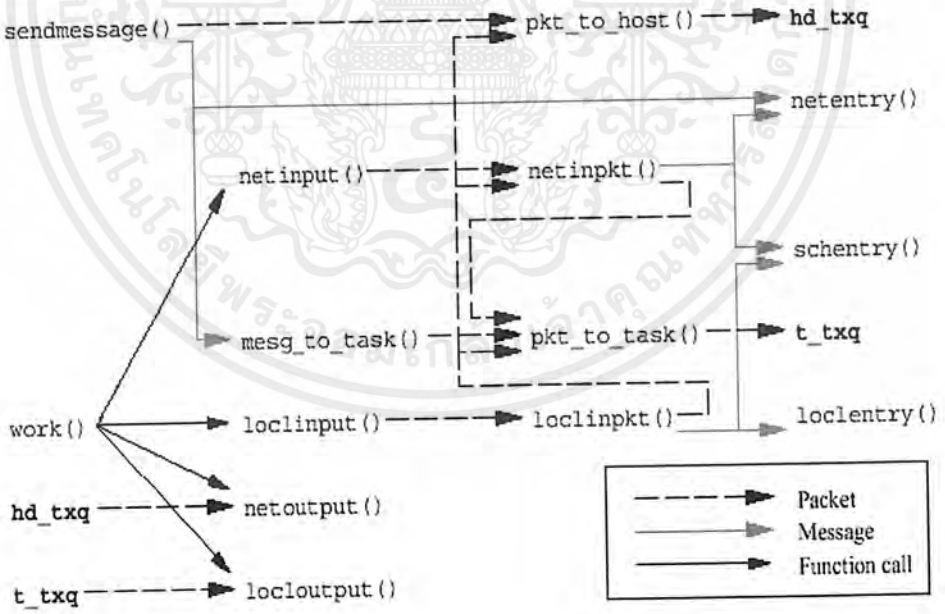
รูปที่ 3.11 Pvmd-task packet header

ไม่ต้องการ squeues number มีเพียง SOM และ EOM โดย TCP ไม่ต้องมีการบันทึกจากหลาย packet มี length ในการส่งไปใน header แต่ละด้านจะรักษา FIFO ของ packet ในการส่งและมี switches ระหว่างการอ่านใน socket และมีการเขียนขณะเป็นที่ว่าง TCP มีการเคลื่อนย้ายไปยังส่วนต่างๆของ packet อย่างไร ถ้าเป็น UDP – sendto(),recvfrom() ที่จำเป็น TCP packet จะส่ง write() ในการเรียกแต่ถ้าได้รับ two read() คือ header และ data เมื่อมีการส่งข้อมูลมากก็จะส่งผลให้ความเหมาะสมของการสื่อสารลดลงโดยต้องการความยาวมากขึ้นของ packet header โดยต้อง gettingจาก packet ทั้งสองคือ header และ packet body จึงจะทำได้สมบูรณ์ แบบ version 3.3 มีการ Unix_domain stream socket แทน TCP สำหรับการเชื่อมต่อ จะพัฒนา latency (ระยะเวลาในการเรียกข้อมูล ส่งข้อมูล) และ tranfer rate (อัตราการถ่ายโอนข้อมูล) ถ้าเป็นไปได้ stream socket ควรใช้ระหว่าง pvmd และ task ดีกว่าระหว่าง task บน host เดียวกัน

3.6 Message Routing

3.6.1 Pvmd

Packet buffer packet descriptor (struct pkt)จะคอยติดตาม message ที่ fragment ทาง pvmd ซึ่งมี fileds pk_buf , pk_mak,pk_dat และ pk_len ที่ใช้ในการ frag ซึ่ง pkt จะบรรจุ state ของการทำงาน pvmd-pvmd โปรโตคอล



รูปที่ 3.12 packet และ message ที่ถูกส่งโดย pvmd

Message Routing เป็นส่วนหนึ่งของ packet ที่ได้รับข้อความจาก network ผ่านทาง netinput() ส่งตรงไปยัง buffer ซึ่งเมื่อ PVM ได้รับจะอ่านด้วย loclinput() ซึ่งจะสร้าง packet ที่เพียงพอ packet จะกำหนดเส้นทางนั้น pvmd เชื่อม queuesกับ เส้นทางที่จะไป ถ้าpacket เป็น muticast descriptor จะจำลองการนับที่พิเศษภายใต้ buffer แล้วทำการ copy แล้วค่อยส่งหลังจากการบันทึกทำให้ buffer ว่าง

3.6.2 Pvmd and Foreign Task

โดยทั่วไปแล้ว pvmd ไม่ต้องการสื่อสารกับ foreign table โดย pvmd จะมี message buffer สำหรับ pvmd อื่นๆ และแต่ละ taskอยู่แล้ว จึงไม่ต้องการ ressembly buffer สำหรับ foreign task ในกรณี task ตาย pvmd ต้องได้รับแจ้งจาก task's pvmd โดยข้อความที่ ressembled ด้วย task'slocal pvmd และได้รับจาก pvmd โดยตรง source address จะถูกรักษาดังนั้นผู้ส่งสามารถ identified Libpvm รักษาข้อมูลใน buffers ดังนั้นจาก pvmd ไปยัง task ไม่ใช่สาเหตุของปัญหา

3.6.3 Libpvm

4 ฟังก์ชันเกี่ยวกับ packet ที่ใช้ในการเข้าออก libpvm

- mrout() ถูกเรียกจาก highter_level function เช่น pvm_send และ pvm_rec ที่บันทึกข้อความเข้าออกจาก task เราจึงจำเป็นต้องกำหนดและสร้างเท่าที่จำเป็นก่อนที่จะเรียก mxfer().mxfer() ซึ่งจะทำการ blocking จนกระทั่งได้รับหรือหมดเวลา
- mxinput() จะบันทึก fragments ไปใน task และ ressembly ข้อความซึ่งทั่วไปของ PVM mxfer()
- select()ในการกำหนดเส้นทางของ socket ให้เป็นตามลำดับในการเข้าออก
- pvmmctl() ถูกเรียกโดย mxinput() เมื่อต้องการควบคุมข้อความที่จะรับ Direct Message Routing

การเชื่อมต่อโดยการกำหนดเส้นทางโดยตรงจะอนุญาตให้แค่หนึ่ง task ส่ง message ไปยัง task อื่นๆผ่านทาง TCP link หลีกเลี่ยง overheadที่เกิดขึ้นกับ pvmd โดยจะมีการทำงานใน libpvm และจะมีการแจ้งล่วงหน้าและควบคุม message ที่จำลองมา By default -> task จะกำหนดเส้นทางไปที่ pvmd ที่ควบคุมมันอยู่ ถ้า direct routingใช้งานได้ (PVM Rout Direct) message จะผ่านเข้ามาที่ mrout() ,ที่มันพยายามสร้างเส้นทางที่ดีที่สุด การกำหนดเส้นทางอนุญาตหรือปฏิเสธโดย destination task หรือ fail message นั้นไปยัง mxfer() Libpvm จะรักษา โปรโตคอลที่ควบคุม block (struct_ttp(b))สำหรับการกระทำที่ปฏิบัติหรือปฏิเสธ การเชื่อมต่ออยู่ใน list ttlist เมื่อต้องการสื่อสาร ,mroute() สร้าง ttpcbและ socket ซึ่งจะส่ง TC_CONREQมาควบคุม message ผ่านทาง default route ในขณะเดียวกันมันจะส่ง TM_NUTIFT ไปที่ pvmd ซึ่งมีการแจ้ง ถ้า task exits ด้วย closure TC_TASKEXITแล้วจึงใส่ ttpcb ใน state TTCONWAITและเรียก mxfer()ใน blocking mode มาจนกระทั่งมันเปลี่ยน state

เมื่อ destination task เข้า mxfer()(เวลาได้รับ message)จะได้รับ TC_CONREQ message ความต้องการจะถูกยอมรับเมื่อสถานะ (pvmroutepot!=PvmDontRoute)และมีการ

ทำงานให้มีการเชื่อมต่อกันโดยตรงซึ่งจะมีทรัพยากรที่หาง่ายนำมาใช้และ โปรโตคอล version (TDPROCOC) ที่เราต้องการใช้มันทำโดยสร้าง ttpch กับ state TTGRNWAIT, สร้างและฟังจาก socket และตอบไปทาง TC_CONACK message ถ้าการเชื่อมต่อไม่ถึงจุดหมายปลายทางขั้นแรก จะได้รับ TC_CONACK message และเชื่อมต่อกับ state (STATE=TTOPEN)แล้ว mrout() ผ่านไปยัง mxfer() แล้วจะส่งเมื่อการเชื่อมต่อที่ถูกปฏิเสธ ป้องกันการกระโดดข้าม ซ้ำไปซ้ำมา ถ้าจุดมุ่งหมายของ TC_CONACK ไม่สูญหาย เพราะมี TC_CONREQ ถูกทิ้งไว้โดยสมบูรณ์ อย่างไรก็ตาม TC_TASKEXIT จะแพร่โดยบอกล่วงหน้าและ ttpeb จะให้ค่า TTDENY โดยการเชื่อมต่อแบบนี้ ทั้งคู่พยายามเชื่อมต่อในเวลาเดียวกันทั้ง TTCONWAIT ซึ่งแต่ละอันจะได้รับ TC_CONREQ message มันจะไปที่ TTOPEN state โดยตรง

3.6.4 Multicasting

Libpvm มีฟังก์ชัน pvm_mcast() ในการส่งข้อความไปที่จุดมุ่งหมายหลายแห่งมันใช้ 1:N fanout ซึ่งแน่นอนมันไม่พอเพียงสำหรับ host แต่ไม่ใช่สาเหตุของการสูญเสียข้อมูล packet ในชั้นของการหาเส้นทางของ pvmd ร่วมกับ libpvm ในการส่ง muticast message โดย mutticast address TID(GID),G bit ,L ถูกกำหนดโดยผู้นับมีค่าเพิ่มขึ้นตาม multicast ดังนั้น multicastใหม่ addressจึงไม่เหมือนเดิม โดยมีการทำงาน task ส่ง TM_MCA message ไปให้ pvmd ประกอบไปด้วยรายการของผู้รับ TIDs ,pvmd จะสร้างmulticast descriptor (struct MCA)และ GID ซึ่งจะส่ง address เคลื่อนย้ายสิ่งแปลกปลอมและจำลองพวกมันไปไว้ใน MCA , แต่ละ destination pvmd จะส่ง DM_MCA ด้วย GIA และจุดหมายอยู่ใน host GIB ส่งกลับไปให้ task ใน TM_MCA message Task จะส่ง muticast ไปให้ pvmd , address ไปยัง GID เมื่องานแต่ละงานมาถึงที่ destination pvmd มันจะทำการบันทึกแต่ละ task packet จะถูกรักษา ดังนั้น multicast address และ data socket ของแต่ละ pvmd ถูกใส่ใน header flags

3.7 Task Enviornment

3.7.1 Environment Variable

PVM เพิ่มสิ่งแวดล้อมและอาจทำให้สนับสนุนบน machine และไม่ขัดต่อหลักการ

3.7.2 Stand Input and Output

แต่ละ spawn task ใน PVM จะมี /dev/null เปิดสำหรับ stdin จากนั้นสืบทอดมาเป็น stdout sinkoutput ใช้ stdout หรือ stdcrr ซึ่งถูกอ่านโดย pvmd ผ่านไปยัง PVM message และส่งไปยัง TID ถ้า TID เท่ากับ 0 (ไม่มี parent) message จะผ่านไปที่ master pvmd ที่ซึ่งจะถูกแสดง error ออกมา

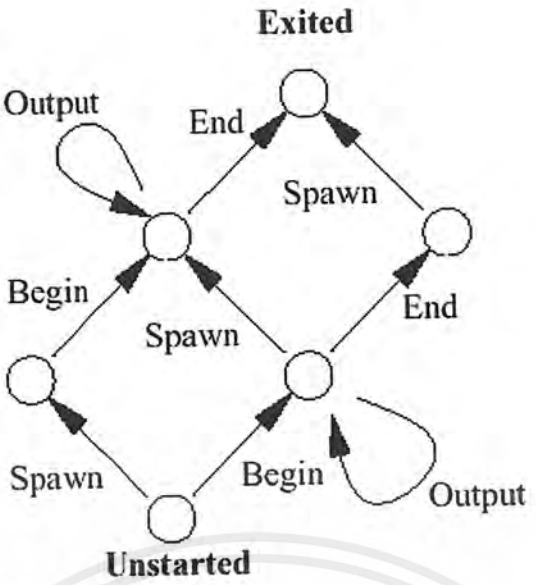
สำหรับงานที่ถูก spawn โดยงานที่สืบทอดจาก stdout sink ก่อนที่จะ spawn parent สามารถใช้ pvm_setopt() แก้ไข output TID หรือ code ที่ไม่มีผลกระทบต่อ output ของ parent งานสามารถกำหนด output TID ได้3 อย่าง คือ ค่าที่ได้สืบทอดจาก parent , ตัวเอง, ศูนย์ มันสามารถจะกำหนด output code ได้ก็ต่อเมื่อ output TID ถูกกำหนดโดยเจ้าของ TID หมายความ

ว่า output นั้นไม่สามารถกำหนดได้โดยอิสระ 4 ชนิด message ที่ถูกส่งไปยัง stdout sink รูปแบบ message สำหรับแต่ละชนิดดังนี้

ตารางที่ 3.5 แสดง message ที่ถูกส่งออกไปยัง stdout

Spawn: (code) {		Task has been spawned
	int tid,	Task id
	int –1,	Signals spawn
	int ptid	TID of parent
}		
Begin: (code) {		First output from task
	int tid,	Task id
	int –2	Signals task creation
	int ptid	TID of parent
}		
Output: (code) {		Output from a task
	int tid,	Task id
	int count,	Length of output fragment
	char data[count]	Output fragment
}		
End: (code) {		Last output from a task
	int tid,	Task id
	int 0	Signals EOF
}		

ขั้นแรก มีสองส่วนใน message คือ task id และ output count ที่ผู้รับสามารถแยกความแตกต่างระหว่างงานและชนิดของ message ได้ สำหรับแต่ละ task หนึ่ง message ของแต่ละชนิด spawn, Begin, End ที่ส่ง ร่วมกับ 0 หรือ messageอื่นๆของ class output (count > 0) class Begin, Output และ End จะได้รับเหมือนมาจากแหล่งเดียวกัน pvmd ของ task เป้าหมาย class spawn เริ่มมาจาก pvmd ของ parent task output sink ถูกคาดหมายว่าจะเข้าใจประเภท message แต่ละ message และทราบว่ามีการหยุดจาก task หรือกลุ่มของ task message ถูกออกแบบเพื่อป้องกันปัญหาเมื่อ task spawn กับ task อื่นๆจะลบออกทันทีโดย output จะใส่ค่า End message จาก task พ่อ และเลิกการทำงานหลังจากได้รับ output จาก task ลูกตามกฎ Spawn messageสำหรับ 2 task จะมาถึงก่อน End messageของtaskแรก the Begin messageจำเป็นเพราะ spawn message มาถึงช้ากว่า End message ในTask เดียวกัน



รูปที่ 3.13 Output states of task

libpvm ฟังก์ชัน pvm_catchout ใช้กับ output ที่รวบรวม output จาก task ลูกไปไว้ใน file โดยจะ set output TID ไว้ในตัวมันของ taskidและ outputcode สำหรับควบคุม message TC_OUTPUTคือ outputจาก task ลูกและ task หลานถูกรวบรวมโดย pvmdและส่งไปที่ pvmmctl และพิมพ์โดย pvmclaimo()

3.7.3 Tracing

Limpvm library มีระบบการบันทึกซึ่งสามารถบันทึกparameterและผลจากการเรียกใช้ interface function Trace data ถูกส่งคล้ายmessageไปยัง Trace sink task (task ที่ต่ำกว่า) และ output ถูกส่งไปที่stdout sink ถ้า Trace output TID เป็น ศูนย์ Tracingนั้น disabled

- รอบๆtrace sink , task มีการสืบทอด trace mask ใช้ tracing function โดยแสดงผ่าน sting ที่ PVMTMASK ซึ่ง task สามารถคำนวณ trace markของมันเองและ taskที่สืบทอดจากมันได้ โดย task'strace mark set ด้วย TC_SETTMASK
- ค่าคงที่เกี่ยวกับtrace message ดูใน file pvmtev.h ซึ่งข้อมูลของ trace data จาก task ถูกรวบรวมส่วนที่เหมือนกันไว้ด้วยกันเช่น TEV_SPNTASK.TEV_NEWTASK และTEV_ENDTASKเป็นtrace message โดยรวบรวมไว้ใน pvmd

3.7.4 Debugging

เป็นส่วนที่ PVM จัดการ เริ่มจาก task จะทำงานภายใต้debugger แต่มีกระบวนการที่ยุ่งยากในการ spawn ผู้ใช้จึงต้องเรียก pvm_spawn() และเริ่มงานใหม่ ถ้า Pvm TaskDebug ถูกผ่านไปยัง pvm_spawn() งานจะเริ่มทำทาง debugger script คือ $ PVM_ROOT/lib/debugger โดย pvmd จะส่งชื่อและ parameter ของ task ไปให้ debugger scrip ซึ่งเป็นจุดเริ่มต้น scriptจัดทางที่ง่าย xtermใน windowมันจะทำงานตามสถาปัตยกรรมของมันบน host scrip สามารถเปลี่ยนแปลงโดยผู้ใช้ pvmd ทำงานบน debugger ที่ต่างกันได้ โดยผ่านทาง bx= host หรือ PVM_DEBUGGER

3.8 Console Program

ใช้ในการควบคุม virtual machine ซึ่งมีหน้าที่เริ่มและหยุด processซึ่งมี libpvm functionที่ถูกสร้างขึ้น Pvm_getfds() และ select()ใช้ในการตรวจสอบ inputจาก keyboard และ message จาก pvmd input ของkeyboardไปยังตัวแปรคำสั่งขณะที่messageจะมีการบอก addหรือoutput จาก task -console รวบรวม outputหรือ trace message จาก spawned task โดยใช้ redirection mechanismsและบันทึกลง fileโดยจะใช้ begin และ end message จากtaskลูกรักษากลุ่มของtask ที่มีการสืบทอดมาจากบรรพบุรุษใช้ PvmhostAdd สำหรับเหตุการณ์ที่ notify โดยแจ้ง user เมื่อ virtual machine ถูกการ reconfigured

3.9 Resource Limitations

Resource limitsถูกกำหนดโดย OS และ Hardware ที่เกี่ยวข้องกับ PVM ถ้าเป็นไปได้ PVMหลีกเลี่ยงการติดตั้ง limitที่แน่นอน โดยจะให้ error กลับมาเมื่อ resource หมด การแข่งขันระหว่าง user บน host เดียวกันหรือ network มีผลกระทบกับ limit dynamically

3.9.1 In the PVM Daemon

แต่ละ task ,pvmd สามารถตั้งค่า limited โดย2 factor

- number of process ที่ผู้ใช้ได้รับอนุญาตจาก OS
- number of file descriptor ที่มาจาก pvmd

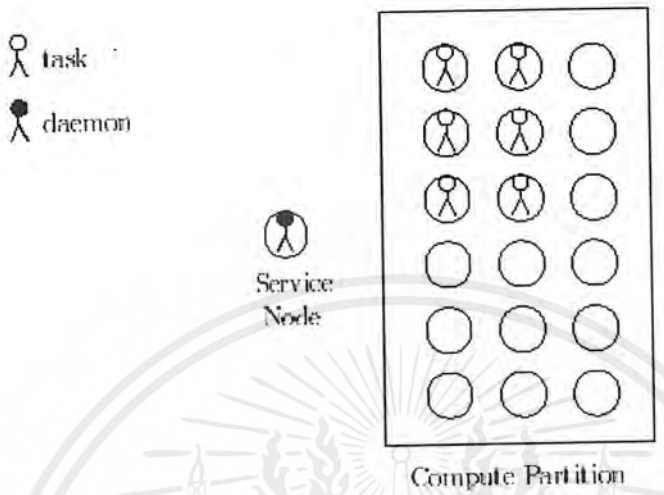
แต่ละ task ใช้ file descriptor ใน pvmd คือ pvmd_task TCP แต่ละการ spawn task ใช้ extra descriptor The pvmd อาจเกิดปัญหาคอขวดถ้า task พยายามติดต่อในหลายทาง The pvmd ใช้ dynamicalในการแยก และรักษา message ใน packet ระหว่าง task และเมื่อได้รับ task มันส่งไปใน pvmd ใน FIFO สรุปแล้ว Task สามารถส่งเมื่อได้รับ end ว่า " off " ถ้าไม่ได้รับ system ออกจาก memory

3.9.2 In the Task

Task สามารถมี number และสามารถเชื่อมต่อกันโดยตรง โดยการเชื่อมต่อกันแต่ละครั้ง task ต้องมีการติดตั้งบน TCP และมีการใช้ File descriptor โดยค่าสูงสุดของ PVM ดูได้จาก memory ของ task เพราะ message จะมีการ packet และdata ใน memory ระหว่างpackและตัวถูกส่ง ในส่วนที่คล้ายกันถ้าหลายๆ task ส่งไปที่เดียวกันที่จุดปลายทางของ task หรือ pvmd ทำให้เกิด overloads โดยมันพยายามเก็บ message ปัญหานี้สามารถหลีกเลี่ยงได้โดยเรียง application code ก่อนการทำงาน

3.10 Multiprocessor System

PVM จะมีการยอมรับและปฏิเสธในการส่ง message_passing programming และใช้โปรแกรมเดียวกันใน mutiprocessor



รูปที่ 3.14 PVM and task on MPP host

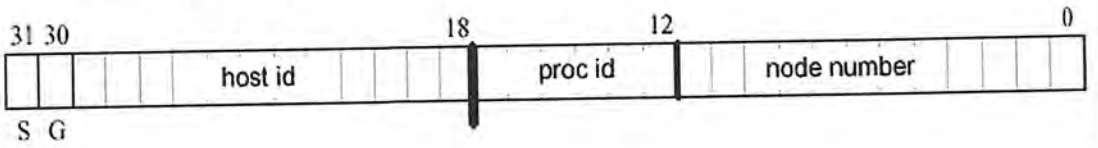
Multiprocessorแบ่งได้ 2 ลักษณะ

- message passing
-shared memory

message passing จะสนับสนุนบน Intel'siPSC/860และ paragon porting PVM ส่วนสอง เคลื่อนย้ายข้อความ buffer ใน shared memory

3.10.1 Message Passing Architectures

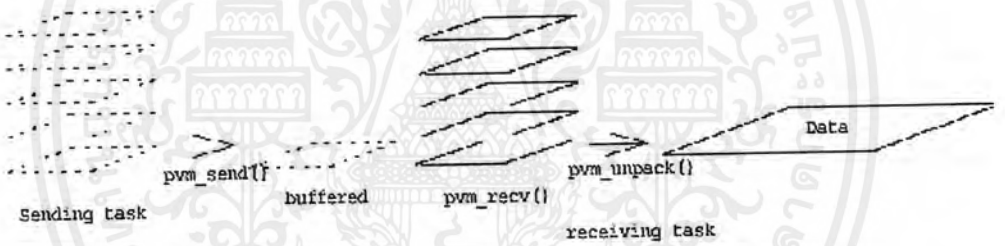
4Task เริ่มจาก node ของ Unix process และเชื่อมต่อกับ TCP socket และ pvm task ผ่าน node ทาง pvmspawn() เมื่อdaemon ได้รับ request ของ task ใหม่ให้ spawn() มันจะจอง node เท่าที่จำเป็นและสามารถเปลี่ยน messageโดยการเปลี่ยน address The NX OS มีการจำกัดจำนวน subcubes ไม่เกิน10 pvm_spawn จะfail เมื่อมีค่าความเป็นไปได้ไม่เพียงพอ PVM message ส่งผ่าน functionของการทำงานในรูปของ native send และรับระบบ call, addressของ taskใช้ในการเข้ารหัส (encode)ใน taskid



รูปที่ 3.15 แสดงการจำแนก TID บน MPP

โดย message สามารถส่งไปยัง task เป้าหมายโดยปราศจากการช่วยเหลือจาก deamon แต่ physical addressใช้ psc/860 สำหรับ intercube การสื่อสาร โดยมีตัวเลขที่คงที่ ใช้โดย task เป้าหมายทำงานบน node เดียวกัน PVM ใช้ asynchronous ในการส่งmessage ซึ่ง OS สามารถส่งได้อย่างรวดเร็วถ้ามีข้อมูลเล็กจำนวนมากหรือข้อมูลใหญ่สำหรับการส่งแต่ละครั้ง โดย PVM จะติดต่อกับ switchในการ synchronous เมื่อไม่มี message หรือ system ของbuffer เสียหายในการพัฒนาประสิทธิภาพ task เรียก pvm_send() เมื่อ task เรียง pvm_recv() message ไปอยู่ใน buffer ซึ่ง pvm buffer เป็น packet ทำงานระหว่าง pvm_send()/pvm_recv()

รูปที่ 3.16 การแตกข้อมูลเป็นข้อมูลเล็กๆขนาดคงที่



รูปที่ 3.17 buffering:การ buffering ข้อมูลส่วนหนึ่งจนกว่า pvm_recv จะถูกเรียก

MPP ถูกกระทำด้วย workstation โดยจะทำงานร่วมกับ pvm libray ที่อยู่บน Unix Socket สำหรับติดต่อ message โดยธรรมดาจะหลีกเลี่ยงการ running บน front end เพราะการเชื่อมต่อระหว่าง front end และ node process ต้องไปทาง pvm deamon และ TCP socket ซึ่งการคำนวณและการสื่อสารทั้งหมดควรจะมี computer node ตามลำดับ การทำงานที่ได้เปรียบและความเร็วในการติดต่อระหว่าง pvm deamon และ TCP socket pvm libray สำหรับ front end จะแตกต่างจากโหนดเดียว การทำงานจะแตกต่างจากคอมพิวเตอร์หลายโหนด

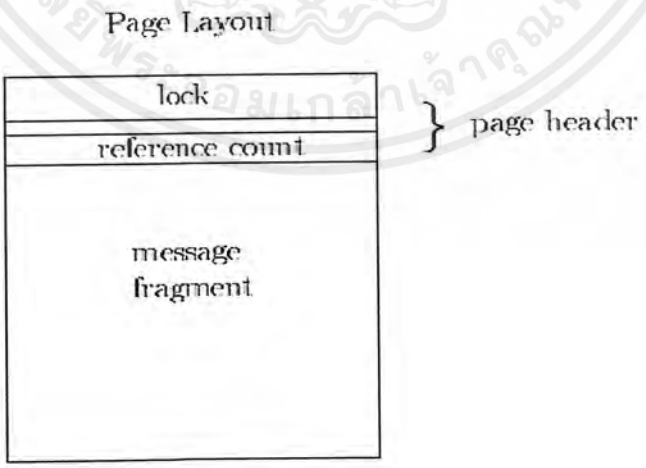
ตารางที่ 3.6 แสดง native system

PVM	iPSC860	Paragon	CM-5
Pvm_spawn	getcube/load	nx-loadve	fork
Pvm_send	isend/csend	isend/csend	CMMD_send_async/_noblock
Pvm_recv	irecv	irecv	CMMD_receive_async
Pvm_mcast	gsendx	gsendx	CMMD_send_async/_noblock

ข้อมูลข้างบนแสดง native system ที่ใช้เรียกโดยตรงตาม PVM function บนหลายๆ platform CM-5 จะแตกต่างจาก Intel เพราะต้องการ host พิเศษสำหรับแต่ละกลุ่มของ task spawn ซึ่งกระบวนการนี้อยู่ใน PVM และมีการส่ง message ช้า ระหว่าง pvmd และ node program จึงจำเป็นต้องมี adds overhead ไปยัง deamon task CM-5 จะถูกจำกัดสำหรับ node ที่อยู่ partition เดียวกัน เมื่อ pvm spawn 2 กลุ่ม ของ task จะมีการ share time ใน partition และการสื่อสารที่ทำบน pvmd CMMD support การทำงานแบบ multicastion ดังนั้น pvm_mcast () จะทำงานด้วย loop ของ CMMD_async_send()

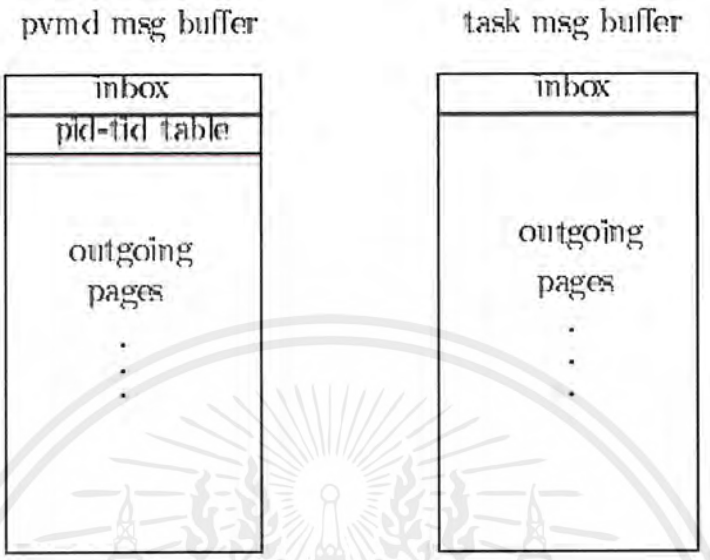
3.10.2 Shared_Memory Architectures

Share memory จัดการเมื่อมีกระบวนการในการแลกเปลี่ยน data ในการทำงาน แต่ละ task จะมีการ share buffer ของตัวมันเองด้วยการสร้าง shmget() task ใช้ id ในการ share segment ถ้ามี key จากผู้ใช้คนอื่น PVM จะกำหนด id ลับ มาให้ task และ task จะสามารถติดต่อกับ task อื่นๆโดยทำการ mapping buffer กับ id ใน memory ที่ว่างของตัวเอง message buffer ถูกแบ่งเป็น page ซึ่งแต่ละส่วนทำการ fragment



รูปที่ 3.18 โครงสร้างของ PVM

ขนาด Pvm's page สามารถเปลี่ยนแปลงได้ แต่ละ page จะมี header ซึ่งบรรจุ the lock และ เลขที่อ้างอิง (reference count) page แรกทำให้เป็น incoming box ขณะ page ที่เหลือจะอยู่ที่ outgoing fragment



รูปที่ 3.19 โครงสร้างของ shared message buffers

ในการส่ง message task แรกจะไปอยู่ใน buffer และจะส่ง message header (sender TID และ location ของข้อมูล) ไปยัง incoming box ของผู้รับ เมื่อ pvm_rec() ถูกเรียก PVM ก็จะ check incoming box และลดพื้นที่ เพื่อแก้ปัญหาความสับสนไม่ให้เกิดขึ้น ถ้าไม่สามารถส่ง header อาจจะมีสาเหตุจาก receiving box นั้นเต็ม มันจะหยุดรอจนกระทั่ง task พร้อม overhead จะเกิดอย่างหลีกเลี่ยงไม่ได้เมื่อมี message ที่ถูก pack และ unpack จาก buffer ใน PVM ถ้า buffer เต็ม ข้อมูลจะถูกบันทึกลงไปใน buffer สำรองในที่ว่างส่วนตัว และจะมีการ share buffer ภายหลัง Memory จะไม่มีปัญหาถ้าแต่ละ process มี buffer ของตัวเอง และแต่ละ page buffer มี lock ของตัวเอง ถ้า page ถูก lock จะไม่มี process ใดพยายามที่จะอ่านมันเพราะ header ห้ามไม่ให้ sign out แต่ถ้ามีการอ่านจาก page เดียวกันมันพยายามส่งข้อความไปให้ header ภายในเวลาเดียวกันแต่อาจจะล่าช้าเพราะ header นั้น short (16 bit) PVM พยายามใช้ small number ของ page ใน message buffer และนำกลับมาใช้ใหม่เมื่อมีการอ่านจากผู้รับแล้ว

3.10.3 Optimized Send and Receive on MPP

User ใช้ message จาก PVM โดย pack data ไว้ใน PVM buffer ก่อนการส่งและ unpack หลังจากได้รับเข้าไปใน buffer แล้วซึ่งการทำงานได้ผลดีกับการสื่อสารเช่น Ethernet บน MPP system packing และ unpacking ทำให้เกิด overhead แก้ปัญหาโดยเพิ่ม 2 PVM function คือ pvm_psend() และ pvm_precv() ซึ่งจะทำการเชื่อม packing/unpacking และ sending/receiving ในขั้นแรก คือจะเชื่อมโดยตรง native message ที่มีประโยชน์บนระบบซึ่งจะทำกับ buffer ภายใน

ด้วยกันเอง บน Paragon ฟังก์ชันใหม่ ให้มีประสิทธิภาพเหมือนกันเช่นเดียวกับ native ones อย่างไรก็ตามผู้ใช้สามารถใช้ทั้ง pvm_psendและ pvm_send() ในโปรแกรมเดียวกันบน MPP pvm_psend ต้องจับรวมกับ pvm_precv() และ pvm_send() กับ pvm_recv()

บทที่ 4
ฟังก์ชันของไลบรารี่ PVM

4.1 ในการใช้ไลบรารี่ของ PVM มีตัวแบบของการเขียนอยู่ 3 ลักษณะคือ

4.1.1 Crowd Computation

เป็น Model ที่กลุ่มของ Process ที่มีความสัมพันธ์กันจะ execute โค้ดของโปรแกรมที่เหมือนกัน แต่ว่าจะทำการคำนวณในส่วนที่ต่างกันของ Workload และจะมีการรวบรวมผลลัพธ์ที่ได้ของแต่ละส่วนมาเป็น Output ซึ่ง Model นี้จะแบ่งได้อีกออกเป็น 2 แบบ คือ

1 Master-Slave เป็น Model ที่จะมีการแบ่งเป็น program ในส่วนที่จะเอาไว้ควบคุมซึ่งจะเรียกว่า "Master" (host node) จะควบคุมในส่วนการเริ่มต้น,การรวบรวมผลลัพธ์และการแสดงผล และอีกส่วนหนึ่งคือ "Slave" program ซึ่งจะเป็นส่วนที่จะทำการคำนวณจริง โดยจะได้รับข้อมูลการทำงานจาก ส่วนของ Master ที่จัดแบ่งให้

2 Node-Only เป็น Model ที่จะมีโปรแกรมที่ทำงานเพียงโปรแกรมเดียว โดย node ต่างๆจะรับรู้เองว่าจะต้องทำงานในส่วนไหนของโปรแกรมโดย model แบบ Crowd Computation นี้จะแบ่งเป็น 3 Phases

- ส่วนแรก เป็นส่วนของการเริ่มต้นของ parameter ต่าง และกระบวนการจัดแบ่ง Process, จัดแบ่ง Workload
- ส่วนที่สอง เป็นส่วนของการคำนวณ
- ส่วนที่สาม เป็นส่วนของรวบรวมผลลัพธ์และแสดง output และทำการทำลาย process ทิ้ง

4.1.2 Tree Computation

เป็น Model ที่ process ต่างๆจะเริ่มต้นและมีการสร้าง process ขึ้นมาแบบไม่แน่นอน (Dynamic) model แบบนี้จะมีการสร้าง process แบบ tree หรือ parent-child (ตรงกันข้ามกับ Crowd ที่มีรูปแบบคล้าย star) model แบบนี้มักใช้กันใน program ที่ไม่ทราบจำนวน workload ที่แน่นอน เช่น "divide and conquer" , recursive, branch-and-bound algorithm

4.1.3 Hybrid Computation

เป็น Model ที่รวมเอาทั้ง Crowd Computation และ Tree Computation เข้าด้วยกัน

4.2 การจัดแบ่งงาน

มีหลักการ 2 แบบที่ใช้ในการจัดแบ่ง Workload คือ

1. **Data decomposition** เป็นการแตกข้อมูลออกเป็นส่วนย่อยแล้วทำการคำนวณข้อมูลในลักษณะขนานในส่วนย่อยๆนั้น จากนั้นก็นำเอาผลลัพธ์ที่ได้จากการคำนวณในแต่ละส่วน มารวมเป็นผลลัพธ์ที่สมบูรณ์

2. **Function decomposition** เป็นการแตก program หรืองาน ออกเป็น ส่วนการทำงานย่อย หรือ เป็น function ย่อยแล้วทำงานในส่วนของส่วนการทำงานย่อยหรือ function ย่อย นั้นในลักษณะ Pipeline คือทำงานไปด้วยกัน พร้อมๆกัน จากนั้นก็นำผลลัพธ์ที่ได้ ส่วนของงานหนึ่งมาเป็น input ของส่วนของงานอีกอันหนึ่งมารวมเป็น

4.3 การเรียกใช้ฟังก์ชัน PVM

PVM จะใช้รูปแบบของ ภาษา C และ Fortran ซึ่ง Library ของ PVM จะ link รวมกับ compiler ภาษา C และ Fortran เราสามารถเขียนโปรแกรมแบบ C หรือ Fortran ก็ได้แล้ว compile ด้วย คำสั่ง "aimk" ของ PVM ซึ่งจะตรวจสอบรูปแบบของโปรแกรมที่เขียนและ link ไปยัง compiler ตามรูปแบบโปรแกรมที่เขียนนั้นโดยอัตโนมัติ

PVM จะมีการให้หมายเลขของแต่ละงาน คือ TID กับงานทุกๆ งาน โดย ตัว PVM จะเป็นตัวกำหนดให้ ผู้เขียนโปรแกรมไม่สามารถกำหนดเองได้

การสื่อสารใน PVM Model จะสมมติว่างานแต่ละงานสามารถที่จะติดต่อกันได้และไม่จำกัดจำนวน message ที่แต่ละงานจะใช้ติดต่อกัน การสื่อสารใน PVM Model สามารถที่จะใช้การสื่อสารแบบ Broadcast และ Multicast ไปยังกลุ่มของงานหรืองานทั้งหมดได้ การรับส่งของ message ระหว่างงานจะต้องมีการการใช้ Buffer ซึ่งก็จะเป็นส่วนหนึ่งของ Physical Memory ของเครื่องที่งานนั้นๆทำงานอยู่ โดย Message Buffer จะมีการจัดแบ่งแบบ Dynamic ดังนั้น ขนาดของ Message ที่จะส่งจากงาน A ไปยังงาน B ก็จำกัดด้วยขนาดของ Physical Memory ของ B

PVM Model จะรับประกันการมาถึงของ message ว่าถ้างานที่ 1 ส่ง message A ไปยังงานที่ ที่ 2 และจากนั้น งานที่ 1 ก็ส่ง message B ไปยังงานที่ 2 message A จะต้องไปถึง งานที่ 2 ก่อน message B

4.3.1 ฟังก์ชันเกี่ยวการควบคุมโพรเซส

```
int tid = pvm_mytid(void)
call pvmfmytid(tid)
```

pvm_mytid() จะส่งค่า TID ของงานและสามารถเรียกใช้ได้หลายครั้ง

```
int info = pvm_exit(void)
call pvmfexit(info)
```

pvm_exit() เป็น ฟังก์ชัน ที่จะบอกให้ pvmd ของงานนั้น (local pvm daemon) ว่าให้เอางานหรือ process นั้นออกจาก PVM ซึ่ง ฟังก์ชันนี้ไม่ได้ทำลายงานนั้นทิ้ง งานนั้นสามารถที่จะเริ่มที่จะทำงานใหม่ได้ ซึ่งจะเหมือนลักษณะงานใน UNIX

```
int numt = pvm_spawn( char *task, char **argv, int flagchar
                                *where, int ntask, int *tids)
call pvmfspawn ( task, flag, where, ntask, tids, numt)
```

pvm_spawn จะเริ่มทำงานจำนวน **ntask** ที่จะ execute file **task** บน virtual machine **argv** เป็น Pointer ที่ชี้ที่ array ของ argument ของ **task** ถ้างานไม่มี argument **argv** จะเป็น NULL สำหรับ flag argument จะใช้เป็นการบอก Option ซึ่งจะมี option ดังนี้

Value	Option	Meaning
0	PvmTaskDefault	PVM จะเลือกที่จะ spawn process
1	PvmTaskHost	**where** argument เป็น parameter ที่จะบอกว่าให้ spawn process ที่ไหนบน Host
2	PvmTaskArch	**where** argument เป็น PVM_ARCH(สถาปัตยกรรม) ที่ spawn process ทำงานอยู่
4	PvmTaskDebug	เริ่มงานภายใต้ debugger
8	PvmTaskTrace	เฝ้าดู data ที่ได้ออกมา
16	PvmMppFront	เริ่มงานบน MPP front-end
32	PvmHostCompl	เริ่มงานที่บน host ที่ set ไว้ใน **where**

และจะส่งค่ากลับ numt ซึ่งเป็นจำนวนงานที่มีการ spawn (กำเนิด,เริ่ม) สำเร็จ หรือจะให้ค่า error code ถ้ามีงานที่ไม่สามารถเริ่มได้ ถ้างานสามารถเริ่มต้นได้ pvm_spawn() จะส่งค่ากลับที่เป็น vector ของงานที่เริ่มต้นไว้ใน tids และถ้ามีงานที่ไม่สามารถเริ่มได้ error code จะถูกส่งไปที่ ntask

```
int info = pvm_kill( int tid)
call pvmfkill(tid, info
```

pvm_kill() จะทำลายงานของ PVMที่ถูกระบุไว้ใน tid และฟังก์ชันนี้ไม่ได้ถูก ออกแบบมาให้ทำลายงานของตัวเองที่เรียกใช้ฟังก์ชันนี้

4.3.2 ฟังก์ชันเกี่ยวข้อมูล

```
int tid = pvm_parent(void)
call pvmfparent(tid)
```

pvm_parent() จะส่งค่าของหมายเลขของงานหรือ process ที่เป็นตัวที่สร้างหรือเริ่มงานนี้กลับมาที่ **tid**

```
int dtid = pvm_tidtohost(  int tid )
call pvmftidtohost( tid, dtid)
```

pvm_tidtohost จะส่งค่าหมายเลขของ daemon ที่ run งานหมายเลข tid อยู่กลับมาที่ตัวแปร **dtid**

```
int info = pvm_config( int *nhost, int *narch,
                       struct pvmhostinfo **hostp )
call pvmfconfig( nhost, narch, dtid, name, arch, speed, info )
```

pvm_config() จะส่งค่าของ ข้อมูลเกี่ยวกับ virtual machine กลับมาโดยจะส่งค่าจำนวน host ใน virtual machine กลับมาที่ **nhost** และส่งค่าจำนวนที่แตกต่างกันของ data type ใน virtual machine กลับมาที่ narch และ hostp จะเป็น pointer ที่ชี้ array ของโครงสร้างในตัวแปร**pvmhostinfo** ในการ return ค่ากลับแต่ละครั้งแต่ละ **pvmhostinfo** จะประกอบด้วย หมายเลขของ pvmd , host name, ชื่อของ สถาปัตยกรรม, CPU speed ของแต่ละ host

```
int info = pvm_task( int which, int *ntask,
                     struct pvmtaskinfo **taskp )
call pvmftasks( which, ntask, tid, ptid, dtid,
                flag, aout, info )
```

pvm_task() จะส่งข้อมูลเกี่ยวกับ pvm task ที่ run บน virtual กลับมา โดย **which** จะเป็น ตัวแปรที่จะบอกว่างานไหนเป็นงานที่บอกข้อมูลมา ถ้าเป็น 0 จะหมายถึง งานทั้งหมด จำนวนของ task จะถูก return กลับมาที่ **ntask** และ **taskp** เป็น pointer ที่ชี้ไปยัง array ของ struct -pvmtaskinfo โดย array จะมีขนาด **ntask** แต่ละ struct pvmtaskinfo จะประกอบด้วย หมายเลขของงาน, หมายเลขของ pvmd ที่ run งานนี้, หมายเลขของ parent ของงานนี้และ status flag และ file name ที่สร้างงานนี้

4.3.3 ฟังก์ชันการตั้งค่าแบบปรับเปลี่ยนได้

```
int info = pvm_addhosts( char **hosts, int nhost, int *infos )
int info = pvm_delhosts( char **hosts, int nhost, int *infos )
call pvmfaddhost( host, info )
call pvmfdelhost( host, info )
```

ใน C ฟังก์ชัน จะใช้เพิ่มหรือลบ กลุ่มของ hostได้ ใน virtual machine แต่ในฟังก์ชันของ fortran จะสามารถเพิ่มหรือลบ host ได้ทีละ host เท่านั้น ใน Fortran ฟังก์ชันจะส่งค่ากลับมาที่ **info** เป็น status code ใน C ฟังก์ชัน ตัวแปร info จะถูก return กลับมาเป็นจำนวน host ที่เพิ่มหรือลบ สำเร็จ สำหรับ argument **infos** จะเป็น array ที่มีความยาวเท่ากับ **nhost** ซึ่งจะมี status code ของแต่ละ host ที่ถูกเพิ่มหรือลบออก

ฟังก์ชันเหล่านี้ถูกใช้ในการ setup virtual machine ซึ่งใช้ในการเพิ่มความยืดหยุ่นและใช้ในการทำเพื่อรองรับความผิดพลาดที่เกิดได้ (fault tolerance) คือสามารถที่จะลบ host ที่ fail ออกได้สำหรับใน application ที่มีขนาดใหญ่

4.3.4 ฟังก์ชันเกี่ยวการกำหนดค่า option

```
int oldval = pvm_setopt( int what, int val )
int val = pvm_getopt( int what )
call pvmfsetopt( what, val, oldval )
call pvmfgetopt( what, val )
```

pvm_setopt() เป็น function ที่จะสามารถให้ user set หรือ get option ในระบบ PVM ได้ โดย pvm_setopt จะใช้ในการ set option และจะ return ค่าของ option เดิมก่อนที่จะมีการset option ใหม่นี้กลับมาและ option ที่มีให้ set คือ

Option Value		Meaning
PvmRoute	1	นโยบายการค้นหาเส้นทาง
PvmDebugMask	2	debugmask
PvmAutoErr	3	รายงาน error โดยอัตโนมัติ
PvmOutputTid	4	stdout destination for children
PvmOutputCode	5	output msgtag
PvmTraceTid	6	trace destination for children
PvmTraceCode	7	trace msgtag
PvmFragSize	8	message fragment size
PvmResvTids	9	allow message toreserved tags and tids
PvmSelfOutputTid	10	stdout destination for self
PvmSelfOutputCode	11	output msgtag
PvmSelfTraceTid	12	trace destination for self
PvmSelfTraceCode	13	trace msgtag

ส่วนใหญ่ฟังก์ชันนี้จะไม่ค่อยได้ใช้งานกัน

4.3.5 ฟังก์ชันเกี่ยวกับการส่งข้อมูล

การส่ง Message จะประกอบด้วย 3 ขั้นตอนในการทำงาน ขั้นแรกคือ การจัดเตรียม buffer ของข้อมูลที่จะส่งการ โดยการเรียก ฟังก์ชัน pvm_initsend() หรือ pvm_mkbuf() ขั้นที่สองคือเป็นกระบวนการ pack ใส่ใน buffer โดยการเรียก ฟังก์ชัน pvm_pk() ขั้นตอนที่สาม เป็นกระบวนการส่ง message ไปยัง process หรืองานอื่น

การรับ message จะรับแล้วทำการ unpack message จาก buffer ที่รับ message โดย ฟังก์ชันที่ใช้ในการรับ message สามารถ set ให้รับทุก message หรือรับเฉพาะ message ที่มาจาก host ใด host หนึ่งก็ได้ การรับ message จะใช้ฟังก์ชัน pvm_recvf()

4.3.6 ฟังก์ชันเกี่ยวการเก็บข้อมูล

```
int bufid = pvm_initsend( int encoding )
call pvmfinitsend( encoding, bufid )
```

ฟังก์ชันนี้จะถูกใช้ในการจัดเตรียม buffer สำหรับการส่งข้อมูล โดยจะถูกเรียกก่อนที่จะมีการ pack message เข้าไปใน buffer ฟังก์ชัน pvm_initsend() จะทำการลบ buffer ที่จะทำการส่งและสร้าง buffer อันใหม่สำหรับการส่ง message ใหม่ ส่วน encoding จะเป็น รูปแบบของการ pack message โดยจะมีรูปแบบคือ

PvmDataDefault เป็น encoding message ตามมาตราฐานที่กำหนดไว้

PvmDataRaw จะไม่มีการ encoding message จะถูกส่งตาม format ที่ message มาเลย ถ้า host ที่ได้รับ message แล้วไม่สามารถอ่าน message นั้นได้ ก็จะส่ง error code กลับมา

PvmDataInPlace Data จะถูกบันทึกไว้ที่เนื้อที่ส่วนหนึ่ง ใน buffer จะมีเพียงขนาดและ pointer ที่ชี้ไปยัง data ที่จะส่งที่ถูกบันทึกไว้ เมื่อ pvm_send() ถูกเรียก data ที่จะส่งจะถูก copy จากที่ที่บันทึกไว้ส่งไป โดยการทำอย่างนี้จะช่วยไม่ให้ user แก้ไข data ขณะที่กำลังจะถูก pack หรือขณะที่กำลังจะส่ง

```
int bufid = pvm_mkbuf( int encoding )
call pvmfmkbuf( encoding, bufid )
```

pvm_mkbuf() จะใช้ในการสร้าง buffer ที่จะใช้ในการส่ง message ใหม่โดยกำหนดวิธีการ encode message ด้วย โดยมันจะ return ค่า **bufid** ซึ่งเป็น หมายเลขของ buffer กลับมา

```
int info = pvm_freebuf( int bufid )
call pvmffreebuf( bufid, info )
```

pvm_freebuf() ใช้ในการทำลายหรือกำจัด buffer โดยจะทำลาย buffer ที่มีหมายเลขตามที่ระบุไว้ใน bufid โดยควรจะทำหลังจาก message ถูกส่งแล้วหรือไม่ต้องการใช้ buffer แล้ว

```
int bufid = pvm_getsbuf( void )
call pvmfgetsbuf( bufid )
int bufid = pvm_getrbuf( void )
call pvmfgetrbuf( bufid )
```

pvm_getsbuf() จะส่งค่า หมายเลขของ buffer ที่ใช้ในการส่งที่ active อยู่กลับมาที่ตัวแปร bufid สำหรับ pvm_getrbuf() จะส่งค่าหมายเลขของ buffer ที่ใช้ในการรับที่ active อยู่ message กลับมาที่ตัวแปร **bufid**

```
int oldbuf = pvm_setsbuf( int bufid )
call pvmfsetsbuf( bufid, oldbuf )
int oldbuf = pvm_setrbuf( int bufid )
call pvmfsetrbuf( bufid, oldbuf )
```

ฟังก์ชันใช้ในการ set active ให้ buffer ที่ใช้ในการส่งหรือ buffer ที่ใช้ในการรับตามที่ระบุไว้โดยตัวแปร bufid และ return หมายเลขของ active buffer ก่อนหน้านี้กลับมา

ถ้า bufid ถูก set ให้เป็น 0 จะหมายความว่าให้ save data ที่อยู่ใน active buffer ปัจจุบันและกำหนดให้ทุก buffer ไม่มีการ active

4.3.7 ฟังก์ชันเกี่ยวการเตรียมข้อมูลไว้ส่ง

ใน C ฟังก์ชันการ pack data สามารถถูก pack เป็นตาม datatype ที่กำหนด เราสามารถเรียกให้มีการ pack message ได้หลายครั้งใน message เดียวกัน ดังนั้นเราสามารถมี message ที่ถูก pack แล้วได้หลาย datatype

argument แรกของแต่ละฟังก์ชัน เป็น pointer ที่ชี้ไปที่ item แรกของ data ที่จะ pack และ nitem เป็น จำนวน item ที่จะ pack สำหรับ stride จะใช้ในการ packing ซึ่งจะใช้เป็น option ในการ pack ฟังก์ชันต่างๆมีดังนี้

```
int info = pvm_pkbyte( char *cp, int nitem, int stride )
int info = pvm_pkcplx( float *xp, int nitem, int stride )
int info = pvm_pkdcplx( double *zp, int niem, int stride )
int info = pvm_pkdouble( double *dp, int nitem, int stride )
int info = pvm_pkfloat( float *fp, int nitem, int stride )
int info = pvm_pkint( int *np, int nitem, int stride )
int info = pvm_pklong( long *np, int nitem, int stride )
int info = pvm_pkshort( short *np, int nitem, int stride )
int info = pvm_str( char *p )
```

```
int info = pvm_packf( const char *fmt, .... )
```

ฟังก์ชันของการ packing data จะเหมือนกันการ printf ใน C คือบอกว่าจะ pack data ให้อยู่ในรูปแบบไหน

4.3.8 ฟังก์ชันเกี่ยวการส่งและรับข้อมูล

```
int info = pvm_send( int tid, int msgtag )
call pvmfsend( tid, msgtag, info )
int info = pvm_mcast( int *tids, int ntask, int msgtag )
call pvmfcast( ntask, tids, msgtag, info )
```

pvm_send() นี้ message ที่มี label ตรงกับที่ระบุไว้ใน msgtag จะถูกส่งไปยัง process หรือ งานที่ระบุไว้ที่ tid

pvm_mcast() message ที่มี label ตรงกับที่ระบุไว้ที่ msgtag จะถูก broatcast ไปยังทุก process ที่ระบุไว้ใน array ที่ tids ชี้อยู่(ยกเว้น precess ที่เป็นตัวเรียก) และ tids จะมีขนาดความยาวเท่ากับ ntask

```
int info = pvm_psend( int tid, int msgtag,
                      void *vp, int cnt, int type )
call pvmfpsend( tid, msgtag, xp, cnt, type, info )
```

pvm_psend() จะ pack และส่ง array ที่มี datatype ตามที่ระบุไว้ไปยัง process หรืองานที่ระบุไว้ที่ tid โดย type ใน C มีดังนี้

PVM_STR	PVM_FLOAT
PVMBYTE	PVM_CPLX
PVM_SHORT	PVM_DOUBLE
PVM_INT	PVM_DCPLX
PVM_LONG	PVM_UINT
PVM_USHORT	PVM_ULONG

ใน PVM จะประกอบด้วยฟังก์ชัน ที่จะใช้ในการรับ message หลายฟังก์ชัน และจะต้องมีการใช้งานที่เหมาะสมกับฟังก์ชัน ที่ใช้ในการส่งใช้ ฟังก์ชัน pvm_psend() ในการรับก็จะต้องใช้ฟังก์ชัน pvm_precv()

```
int bufid = pvm_recv( int tid, int msgtag )
call pvmfrecv( tid, msgtag, bufid )
```

ฟังก์ชันนี้จะคอย message จนกระทั่ง message ที่มี label ตามที่ระบุไว้ที่ **msgtag** ได้รับจาก งานที่มีหมายเลขตรงกับตัวแปร **tid** ถ้าค่าของ msgtag มีค่าเท่ากับ –1 หรือ 0 จะหมายถึงรับทุก message ที่ได้รับ

```
int bufid = pvm_nrecv( int tid, int msgtag )
call pvmfnrecv( tid, msgtag, bufid )
```

ฟังก์ชันนี้ ถ้า message ที่ต้องการยังมาไม่ถึง มันจะไม่รอรับ และจะ return 0 มาใน **bufid** และฟังก์ชันนี้จะสามารถถูกเรียกได้หลายครั้งเพื่อตรวจสอบว่าได้รับ message นั้นหรือยังในขณะที่ยังทำงานอยู่และเมื่อไม่มีการทำงานแล้ว pvm_recv() สามารถถูกเรียกได้เพื่อมารับ message เดียวกันนี้

ถ้า message มาถึงแล้ว pvm_nrecv() จะเก็บ message ไว้ที่ active buffer ที่ใช้ในการรับที่สร้างขึ้นมาใหม่ แล้วสำหรับ active buffer เดิม จะถูกเก็บค่าไว้โดยการเรียก pvm_setrbuf()

```
int bufid = pvm_probe( int tid, int msgtag )
call pvmfprobe( tid, msgtag, bufid )
```

สำหรับฟังก์ชัน นี้ ถ้า message ยังไม่ได้รับ pvm_probe() จะส่งค่าของ 0 ให้ **bufid** หรือส่งค่าของ id ของ buffer ที่ใช้รับ message นี้กลับ ฟังก์ชันนี้สามารถเรียกได้หลายครั้งในขณะที่กำลังทำงานอยู่เพื่อใช้ตรวจสอบว่าได้รับ message หรือยัง

```
int bufid = pvm_trecv( int tid, int msgtag, struct timeval *tmout )
call pvmftrecv( tid, msgtag, sec, usec, bufid )
```

ในบางกรณีอาจไม่ได้รับ message เลยเนื่องจาก host fail หรือเกิด error ขึ้น ซึ่ง user สามารถใช้ฟังก์ชันนี้ โดยฟังก์ชันจะมีการทำงานเหมือนกับ pvm_nrecv() แต่จะสามารถที่จะกำหนดเวลาในการรอรับ message ได้

```
int info = pvm_bufinfo( int bufid, int *byte, int *msgtag, int *tid )
call pvmfbufinfo( bufid, byte, msgtag, tid, info )
```

pvm_bufinfo() จะส่งกลับ message ที่มี label ตรงกับตัวแปร **msgtag** , source **TID** และ ความยาวของ message ที่ถูกระบุใน bufid

```
int info = pvm_precv( int tid, int msgtag, void *vp, int cnt,
                      int type, int *rtid, int *rtag, int *rcnt )
call pvmf precv( tid, msgtag, xp, cnt, type, rtid, rtag, rcnt, info )
```

pvm_precv() จะรวมฟังก์ชันของการรับ message แบบ block และ การ unpacking ของ buffer ในด้านรับ ซึ่ง ฟังก์ชันนี้มักไม่ค่อยได้ใช้กัน

4.3.9 ฟังก์ชันเกี่ยวการขยายข้อมูลที่ได้รับ

ใน C ฟังก์ชัน สำหรับการ unpacking ข้อมูล จะทำการ unpack จะ buffer ที่ใช้รับ เราจะต้อง match ฟังก์ชัน ที่ใช้ในการรับให้เหมาะสมกับฟังก์ชัน ที่ใช้ในการส่ง **nitem** เป็น จำนวนของ item ที่จะให้ unpack

```
int info = pvm_upkbyte( char *cp, int nitem, int stride )
int info = pvm_upkcplx( float *xp, int nitem, int stride )
int info = pvm_upkdcplx( double *zp, int niem, int stride )
int info = pvm_upkdouble( double *dp, int nitem, int stride )
int info = pvm_upkfloat( float *fp, int nitem, int stride )
int info = pvm_upkint( int *np, int nitem, int stride )
int info = pvm_upklong( long *np, int nitem, int stride )
int info = pvm_upkshort( short *np, int nitem, int stride )
int info = pvm_ustr( char *p )
int info = pvm_unpackf( const char *fmt, .... )
```

pvm_unpackf() จะเหมือน printf โดยจะ unpack ข้อมูลตาม format ที่ระบุไว้

4.3.10 ฟังก์ชันเกี่ยวกับการจัดกลุ่มการประมวลผล

Dynamic process group เป็น ฟังก์ชันที่อยู่ใน PVM โดย PVM จะไม่แสดงการทำงานของ group เอง user จะต้องแสดงที่ทำงานเอง การจัดการเกี่ยวกับ group จะถูกจัดการโดย group server ซึ่งจะถูก start โดยอัตโนมัติเมื่อมีการสร้าง group ขึ้น

PVM task สามารถที่จะ join หรือ leave จาก group ใดๆ ก็ได้โดยไม่ต้องบอกกับ task อื่นล่วงหน้า งานใดๆ สามารถที่จะ broadcast message ไปยัง group ใดๆ ก็ได้

```
int inum = pvm_joingroup( char *group )
int info = pvm_lvgroup( char *group )
call pvmfjoingroup( group, inum )
call pvmflvgroup( group, info )
```

ฟังก์ชันเหล่านี้ใช้ในการ join หรือ leave ออกจาก group ที่ระบุไว้ใน **group** ในการเรียก pvm_joingroup ครั้งแรกจะเป็นการสร้าง group ขึ้นมา ฟังก์ชันนี้จะส่งค่า instance number ของ process ที่ join เข้าไปใน group ที่กลับมาที่ตัวแปร **inum** โดยจำนวนจะเริ่มตั้งแต่ 0 ถึง จำนวนของ member ของ group –1 และงานงานหนึ่งสามารถที่จะอยู่ได้หลาย group

```
int tid = pvm_gettid( char *group, int inum )
int inum = pvm_getinst( char *group, int tid )
int size = pvm_gsize( char *group )
call pvmfgettid( group, inum, tid )
call pvmfgetinst( group, tid, inum )
call pvmfgsize( group, size )
```

pvm_gettid() จะส่งค่าหมายเลขของ process ตามที่ระบุใน group name และ inum pvm_getinst() จะส่งค่า instance number กลับมาที่ **inum** ส่วน pvm_gsize() จะส่งค่าจำนวนของ member กลับมาที่ตัวแปร size

```
int info = pvm_barrier( char *group, int count )
call pvmfbarrier( group, count, info )
```

ในการเรียก pvm_barrier() process หรืองาน จะหยุดทำงานจนกว่าจำนวน **count** process ใน group จะมีการเรียก pvm_barrier() โดยทั่วไปแล้ว **count** ควรเท่ากับจำนวน member ของ group ซึ่งสามารถกำหนด count ได้เพราะการกำหนด group เป็น dynamic เราไม่สามารถที่จะรู้จำนวน member ใน group ได้แน่นอนและจะเกิด error ขึ้นถ้ามี process ที่ไม่ใช่ member ของ group เรียก pvm_barrier()

```
int info = pvm_bcast( char *group, int msgtag )
call pvmfbcast( group, msgtag, info )
```

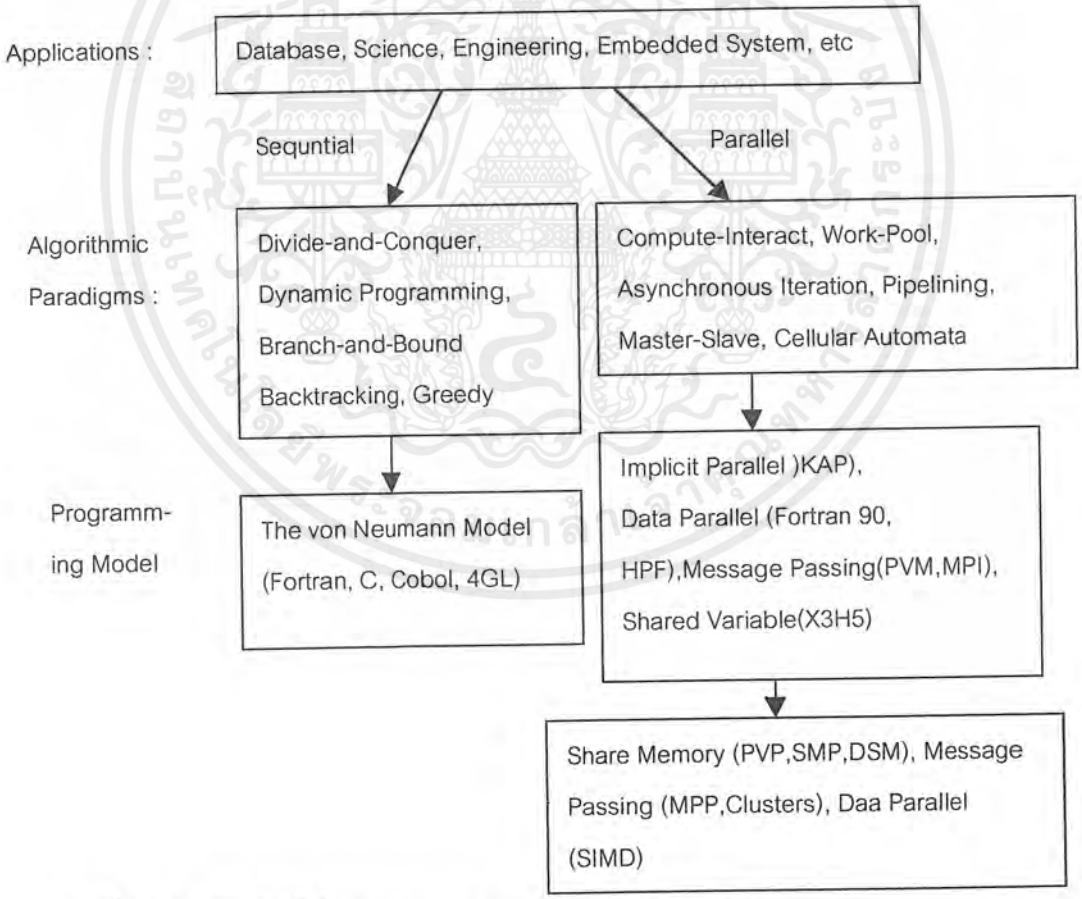
pvm_bcast() จะ broadcast message ที่มี label ตรงกับ **msgtag** ไปยังทุก process ใน group ที่ระบุไว้ใน **group**

บทที่ 5
หลักการเขียนโปรแกรมแบบขนาน

Parallel Programming มีความซับซ้อนและมีการใช้การประมวลผลที่ดีกว่า Sequential Programming โดยมีเหตุผลที่สนับสนุนคือ

- Parallel Programming เป็นการ Programming ที่ทำให้การทำงานหรือการประมวลผลมีประสิทธิภาพมากขึ้นและใช้เวลาน้อยลง
- ทุกวันนี้การ programming แบบ sequential นั้นมีตัวแบบหรือ model ที่ใช้มาจาก basic model เพียง model เดียว (von Neumann Model) ในขณะที่ parallel programming มี model หลายๆ แบบที่แตกต่างกัน
- ซอฟแวร์ เช่น compiler, debugger, profiler ที่ใช้กับ parallel programming มีความซับซ้อนและมีการพัฒนาไปมากกว่าที่ใช้กับ sequential programming

จากรูปเป็นการเปรียบเทียบรูปแบบการ programming ระหว่าง sequential และ parallel



รูปที่ 5.1 เปรียบเทียบโปรแกรมแบบParallel และแบบ Sequential

5.1 Parallel Programming Environment

รูปแบบการทำงานของ Parallel Programming จะเริ่มจากเมื่อ user เขียน code โปรแกรมแบบขนานแล้ว จากนั้น compiler จะแปลง code เหล่านั้นเป็นเป็น native code ที่จะใช้ run บนแต่ละ platform ของเครื่อง ซึ่งจะขึ้นกับว่า compiler ว่าทำงานบนแพลตฟอร์มวินโดว์หรือยูนิกซ์

สำหรับ Run-Time Support เป็นกลุ่มของ routine ย่อยๆ ที่จะถูก load พร้อมกับ user code เพื่อที่จะใช้เริ่มการทำงานของ user code เมื่อมีการ execute user code, ใช้ ลบ data ต่างๆ หลังจาก user code ทำงานเสร็จแล้ว และใช้ในการจัดแบ่งและทำลาย data object ซึ่งส่วนของ Run-Time Support จะถูกเรียกให้ทำงานโดย *library* Library เป็นกลุ่มของ routine ย่อยที่ถูกใช้บ่อยๆ ในการ compile user code

Library จะมีพร้อมกับภาษาที่ใช้ในการเขียนโปรแกรม เช่น ใน C จะมี library "stdio" ที่จะใช้ในการควบคุมและดูแลอุปกรณ์ I/O มาตราฐานของ user program

วิธีการเขียน Parallel Programming

ถ้าต้องการเขียนโปรแกรมแบบ parallel programming สามารถที่จะทำได้โดย

1. Library Subroutine เป็นการเพิ่ม library เข้าไปใน sequential language เพื่อสามารถที่จะทำงานแบบขนานได้
2. New Constructs เป็นการใช้ language ใหม่ที่มีการทำงานแบบ parallel เช่น Fortran 90
3. Compiler Directives เป็น programming language ทั่วไปแต่เพิ่ม comment ที่จะทำให้มีการทำงานแบบขนานได้ เรียกว่า compiler directives(or pragmas)

ในรูปเป็นตัวอย่างของการใช้งานในแต่ละแบบโดย (a) เป็นการเขียนโปรแกรมแบบ sequential ตามธรรมดา (b) เป็นการเขียนโปรแกรมแบบ Library Subroutine (c) เป็นการเขียนโปรแกรมแบบ New Construct (d) เป็นการเขียนโปรแกรมแบบ Compiler Directives

```
for (i=0; i<N; i++) A[i] = b[i] * b[i+1];
for (i=0; i<N; i++) c[i] = A[i] + A[i+1];
```

(a) A sequential code fragment

รูปที่ 5.2 วิธีการเขียนโปรแกรมแบบ Parallel

```
id  = my_process_id();
p = number_of_processes();
for (i=id; i<N; i=i+p) A[i] = b[i] * b[i+1];
barrier();
for (i=id; i<N; i=i+p) c[i] = A[i] + A[i+1];
```

(b) Equivelent parallel code using library routines

```
My_process_id(), number_of_processes(), and barrier();
A(0:N-1) = b(0:N-1) * b(1:N)
C = A(0:N-1) + A(1:N)
```

(c) Equivalent code in Fortran90 using array operations

```
#pragma parallel
#pragma shared(A,b,c)
#pragme local(i)
{
        #pragma pfor iterate (i=0; N ; 1)
                for (i=0; i<N; i++) A[i] = b[i] * b[i+1];
        #pragma synchronize
        #pragma pfor iterate (i=0; N ; 1)
                for (i=0; i<N; i++) c[i] = A[i] + A[i+1];
```

(d) Equivalent code using pragmas in SGI Power C

รูปที่ 5.2 วิธีการเขียนโปรแกรมแบบ Parallel (ต่อ)

ถ้าจะเปรียบเทียบข้อดีและข้อเสียของแต่ละวิธีการจะได้ดังรูป

ตารางที่ 5.1 วิธีการที่นำไปใช้กับการเขียนโปรแกรมแบบขนาน

Approach	Example	Advantages	Disadvantage
Library	Fortran90, Cray Craft	Easy to implement, do not need a new compiler	No compiler check, Difficult to implement,
New Structure	Cray Craft HPF,	Allow compiler check, Analysis, optimization	need a new compiler
Directives	Cray Craft	Between library and new constructs Can be ignored on a sequential platform	

การใช้เพิ่ม Library มีการใช้ที่แพร่หลาย เพราะไม่ต้องการ compiler ตัวใหม่โดยจะ link Library นี้เข้ากับ compiler C ไม่ก็ Fortran ในการ compile แต่เมื่อวิธีการนี้ก็ไม่มีการ optimize user code

ส่วนการใช้ New Construct นั้นมีการทำงานที่ยากเพราะต้องมาทำความเข้าใจการทำงานของ compiler ใหม่ แต่ก็ทำให้ มีการทำงานในส่วนของ optimization และ check analysis เพราะ compiler สนับสนุนการทำงานแบบ parallel โดยตรง

Compiler Directives มีข้อดีคือสามารถทำให้โปรแกรมมีการทำงานแบบ parallel ได้โดยการใส่ comment เข้าไป ดังนั้นตัว user code สามารถที่จะนำมา run บน compiler ของ sequential platform ได้ เพราะ compiler ใน sequential platform จะมองส่วนที่ทำให้ทำงานแบบ parallel เป็น comment ของโปรแกรม

5.2 Processes, Task and Threads

5.2.1 Abstract Process

เราจะนิยามว่า Process **P** จะประกอบด้วย 4 ส่วนคือ (P,C,D,S) โดย P ก็คือโปรแกรม หรือ code , C เป็น control state, D เป็น data state, และ S เป็น status ของ process **P**

Control and Data State

Program ส่วนใหญ่จะอยู่บนพื้นฐานของ imperative machine model โดยมีจุดสำคัญคือ สถานะที่มีการเปลี่ยนแปลงได้ (state updating) imperative program สามารถพิจารณาเหมือนกัน state ของเครื่อง ที่จะเทียบ program กับ สถานะ (state) โดยเริ่มต้นจากสถานะเริ่มต้น (initial state)ไปยัง สถานะอื่นๆ Program จะใช้ 2 กลุ่มของตัวแปร(variables) อย่างแรกเป็น Data variable เป็นตัวแปรที่ประกาศโดย user ที่จะใช้ในการเก็บค่า ตัวแปรอย่างที่สองคือ Control variable เป็นตัวแปรที่จะเก็บข้อมูลควบคุมการทำงานของโปรแกรม ซึ่งจะไม่มีการประกาศใช้งานโดย user

ทุกๆ ช่วงเวลาใดๆ ตัวแปรข้อมูล (Data variable) หรือ ตัวแปรควบคุม (Control variable) ของโปรแกรมจะมีค่าของแต่ละตัวอยู่เป็นคู่ คือ data variable, data value หรือ control variable, control value ซึ่ง กลุ่มของ variable ทั้งสองแบบจะนิยามได้โดยใช้ data state หรือ control state และ program state ณ เวลาใดๆ จะเป็นผลรวมของ data state และ control state ที่เวลาขณะนั้น

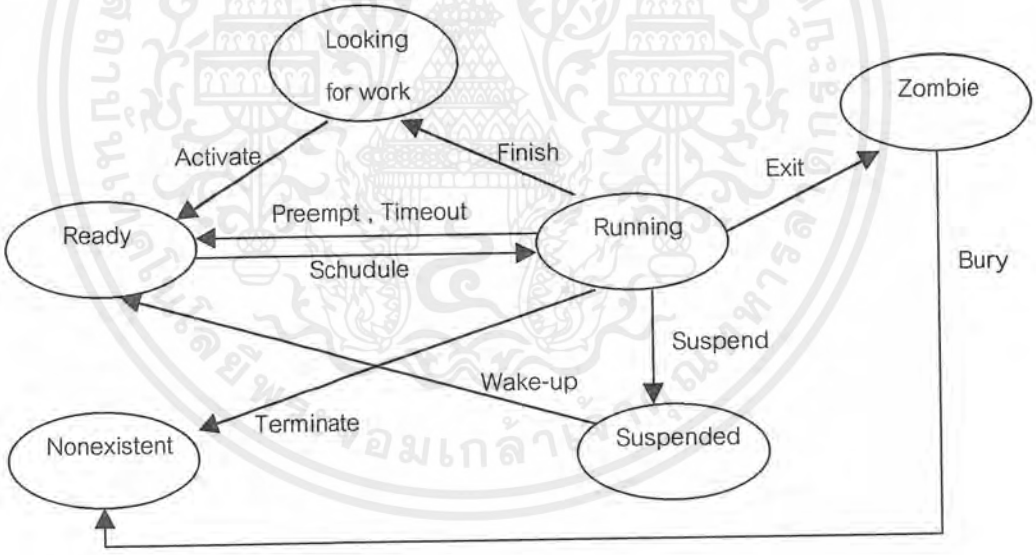
Program จะเริ่มจาก สถานะเริ่มต้น (initial state) เมื่อ program มีการ execute program Program จะเปลี่ยน state ไปจนกระทั่ง program ถึง final state ถ้าโปรแกรมไปสู่ state ที่ไม่สามารถไปยัง final state ได้ จะเรียกว่า program เข้าสู่ "divergent state"

ถ้าจะมองง่ายๆ ให้มอง control variable เป็น program counter สำหรับ process ที่มี thread of control เดียว ก็จะมี control variable ตัวเดียวด้วย ถ้ามี thread of control หลายๆ thread ก็จะมี control variable หลายๆ ตัวด้วย โดยจะมี control variable 1 ตัวต่อ 1 thread สำหรับ subprocess ที่มีหลายๆ subprocess process ทั้งหมดจะมี control variable หลายๆ ตัว โดยจะมี 1 ตัวต่อ 1 subprocess

สำหรับ data variable อาจจะมีการประกาศโดย user หรือโดยตัว compiler เอง ตัวอย่างเช่น data variable ใน UNIX process จะมี hidden data variable ที่ใช้เป็น file pointer ชี้ไปยัง stdout, stdin, stderr

Process status

เป็นสถานะของ process ณ เวลาใดๆ ตัวอย่างเช่นดังรูป



รูปที่ 5.3 Process state transition diagram

* ที่จุดเริ่มต้น process ยังไม่เกิด process จะถูกสร้างโดย parent process หรือถูกสร้างโดย program Process ที่ถูกสร้างขึ้นมาจะอยู่ในสถานะ ready
* process สามารถที่จะพักการทำงานได้โดยจะไปอยู่ในสถานะ suspended เพราะว่า process นั้นไม่สามารถที่จะ run ต่อได้หรือว่าเกิดจากการสั่งของ process อื่น
* หลังจากอยู่ใน state suspended แล้วสามารถที่จะกลับมา run ได้อีกโดยจะไปยัง ready state

* process ที่ run อยู่สามารถที่จะยกเลิกการทำงานไปอยู่ใน ready state อีกครั้งได้ โดย process อื่นที่มี priority ได้หรืออาจจะหมดเวลาที่ให้ใช้ CPU (timeout) ก็จะกลับไปสู่ ready state อีกครั้งเพื่อรอการ run อีกครั้ง

* สุดท้าย process สามารถที่จะ terminate ตัวเองได้เมื่อ execute เสร็จหรือเกิดความผิดพลาด (abnormally)

5.2.2 Execute Mode

Operating System ทั่วไปจะมีส่วนของ

- kernel เป็นส่วนสำคัญของ OS ที่จะใช้ในการจัดการ resource ต่างๆ, ควบคุม error(handle exception), ควบคุม process ใน computer เครื่องหนึ่งจะมี kernel ทำงานอยู่ได้หนึ่งอันเท่านั้น
- Shell หรือเรียกว่า command interpreter เป็นส่วนติดต่อกับ user กับ OS เช่น C Shell บน UNIX
- Utilities เป็นส่วนเพิ่มเติมของ OS เช่น compiler, debugger, editor

Computer จะประกอบด้วย 2 execution mode ในการ execute program ในแต่ละ mode จะบ่งบอกโดยดูจาก register เฉพาะ kernel จะ execute ใน kernel mode หรือที่รู้จักกันว่า supervisor mode หรือ system mode kernel process จะ execute ใน kernel mode และจะมี process ที่ถูกสร้างโดย kernel ที่จะใช้ช่วยในการจัดการทรัพยากรของระบบ เช่น swapper process ใน UNIX

ส่วน program อื่นจะ execute ใน user mode จะเรียกว่า user process shell หรือ compiler ก็จะ execute ใน user mode

Mode Switch

process สามารถที่จะเปลี่ยนจาก kernel mode ไปยัง user mode ได้ เริ่มต้นเมื่อ computer ทำงานจะอยู่ใน kernel mode จากนั้นก็จะเริ่มต้นระบบโดยการสร้าง kernel process จากนั้นก็จะส่งการควบคุมไปยัง Shell ซึ่ง เป็น user mode ที่สามารถที่จะสร้าง user process ได้

เมื่อ user process execute จะสามารถเปลี่ยนเป็น kernel mode โดย 2 กรณี

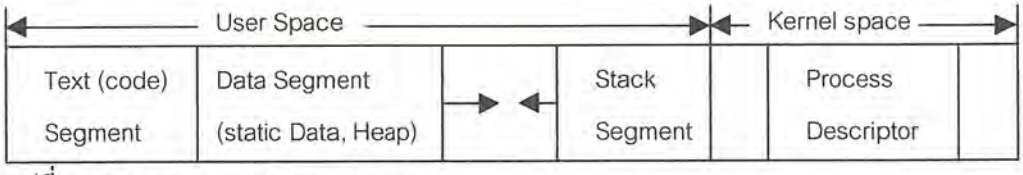
1. เมื่อ user process สร้าง synchronous exception ขึ้น ทำให้มีการเรียก system call ที่ต้องการการบริการจาก kernel
2. เกิด asynchronous exception เช่น disk exception timer

หลังจาก kernel ให้บริการเสร็จแล้ว ก็จะ เปลี่ยน mode ของ process กลับไปยัง user mode อีกครั้ง

5.2.3 Address Space

Computer จะใช้ address ในการอ้างถึง object ต่างๆ โดย address ที่ระบุในคำสั่งจะเป็น virtual address ในขณะที่ address ที่ processor ไปเอาข้อมูลมาใน memory จะเรียกว่า physical memory และขนาดของ address space จะหมายถึงจำนวน address รวมของ computer ที่สามารถอ้างได้

สำหรับรูปแบบ format ของ address space



รูปที่ 5.4 Address space layout of process

โดย text segment จะประกอบด้วย code ของคำสั่งซึ่งมีขนาดคงที่ ส่วน data segment จะเก็บทั้ง static data และ dynamic data ของ process โดยส่วนของ dynamic จะเก็บใน heap และส่วน stack segment จะใช้ในการเก็บข้อมูลในการเรียก procedure

Architecture	**Model**	**Size of Physical Address Space**	**Size of Virtual Address Space**
Intel 80x86	8086	1 MB (2^{32}B)	1 MB(2^{20}B)
	Pentium	4 GB (2^{32}B)	64 TB(2^{20}B)
PowerPC	601	4 GB (2^{32}B)	4 PB(2^{52}B)
	620	16 EB (2^{64}B)	16 EB(2^{64}B)
DEC Alpha	21064	16 GB (2^{34}B)	8 TB(2^{43}B)
	21164	1 TB (2^{40}B)	8 TB(2^{43}B)
MIPS	R4000	64 GB (2^{36}B)	1 TB(2^{40}B)
	R10000	1 TB (2^{40}B)	16 TB(2^{44}B)

รูปที่ 5.5 Address space sizes of popular processors

5.2.4 Process Context

Context ของ process จะเป็นส่วนของ program stateที่จะเก็บใน processor register แต่ register ทุกตัวใน processor ไม่ได้เป็นส่วนของ process context เช่น floating-point register ไม่ได้เป็นส่วนของ context ถ้าไม่ได้ถูกใช้ใน process Context switch จะบันทึก context ปัจจุบันแล้วทำการ load context ใหม่ขึ้นมา โดย context switch จะมีการใช้งานเมื่อมีการเปลี่ยน mode เช่น เมื่อเกิด interrupt ในขณะที่ user process กำลัง execute อยู่ processor จะส่ง การควบคุมไปยัง kernel เพื่อที่จะ execute interrupt handler (interrupt service routine) ก่อนที่ handler จะ execute context ของ user process จะต้องมีการถูกบันทึกไปยัง memory ก่อนจากนั้นก็ทำการลบข้อมูลใน register เพื่อที่จะใช้สำหรับ handler และเมื่อ ทำการ interrupt handler เสร็จแล้ว kernel จะนำ context ของ user process ที่บันทึกไว้ใน memory กลับมาเพื่อที่จะให้ทำงานต่อไป context switch อาจจะต้องใช้เมื่อเกิดการเปลี่ยน state ของ process

5.2.5 Process Descriptor

เป็นข้อมูลของ process ที่ถูกเก็บไว้ในเนื้อที่บางส่วนของ kernel ซึ่งข้อมูลนี่สำคัญมากที่จะทำให้ kernel สามารถที่จะจัดการ process ได้

- Process status จะเก็บข้อมูล state ของ process
- Context จะเก็บ context ของ process ซึ่งพื้นที่ในส่วนนี้จะใช้ในการบันทึก context เมื่อมีการเปลี่ยน state
- Memory map จะเก็บขนาดและ pointer ของ segment และ page table
- Global data structure จะเก็บ pointer ของ queue และ table ที่ kernel จะใช้จัดการ process ทั้งหมดเช่น ทุก process จะเข้า queue เพื่อรอที่จะทำการ execute

5.2.6 Process Control

Process Control จะเกี่ยวข้องกับ function ที่ kernel จะใช้ในการจัดการ process โดย Process Control จะมี function ดังนี้

- Process Descriptors kernel จะใช้ process descriptor ในการสร้าง, suspend, wake up และ terminate process
- Protection kernel จะตรวจสอบว่า process ใช้ทรัพยากรถูกต้องแล้ว process descriptors จะมีข้อมูลเกี่ยวกับสิทธิและข้อมูลของ process ซึ่งโดยปกติ user process จะสามารถที่จะใช้งานในส่วนของพื้นที่ของตัวเองเท่านั้น ไม่สามารถใช้งานพื้นที่ในส่วนของ kernel space หรือ user space ของ process อื่น
- Scheduling จะเกี่ยวข้องกับการจัดทรัพยากรให้ process ต่างๆ โดย kernel program ที่เรียกว่า "scheduler" scheduler จะจัดแบ่งทรัพยากรให้ควรจะจัดอย่างมีประสิทธิภาพและยุติธรรม และจะเป็นไปตาม priority ที่เก็บโดย process descriptors ด้วย

5.2.7 Variations of Process

OS จะให้ address space ของแต่ละ process แยกจากกัน เพื่อความปลอดภัยเมื่อมีการใช้งานแบบ multiuser, multitask อย่างไรก็ดี การจัดการ address space ที่แยกจากกันทำให้เสียเวลาในการจัดการ ตัวอย่างเช่น Unix process execute fork() ระบบ จะทำการสร้าง process ลูกขึ้นมา และaddress space ของ process ลูกก็ต้องถูกสร้างขึ้นมา ก็คือจะมีการจัดแบ่ง memory ให้ และ data segment และ descriptors ของ process พ่อ (parent process) ก็จะต้องถูกคัดลอกใส่ไปใน process ลูกด้วย เราจะเรียก process ใน UNIX ว่า "heavy-weighted process"

กระบวนการสร้างและ switch ของ process ต้องใช้หลาย clock cycle ของ processor ทำให้สิ้นเปลืองทรัพยากรของระบบ จึงมีแนวคิดของ "light-weighted process" หรือที่เรียกว่า thread ความแตกต่างระหว่าง thread และ process คือ thread สามารถมีการ share address space กัน ในการสร้าง thread ก็เพียงแต่สร้าง base thread จากนั้นเมื่อมีการเรียกใช้งาน thread อื่นก็จะสร้าง thread ลูกขึ้นมา และ execute thread ลูกนั้น และจะมีการ share address space กันระหว่าง base thread (parent thread) และ thread ลูก (child thread) ด้วยเพราะมีการจัดแบ่ง memory ที่น้อยและมีการคัดลอก (copy)น้อยครั้ง ทำให้การสร้าง thread เร็วกว่าการสร้าง heavy-weighted process

บทที่ 6
ทฤษฎีของพาราเรลรีซึ่ม

6.1 Homogeneity in Process

ส่วนนี้จะอ้างถึงส่วนประกอบของ process (component process)ที่เหมือนกันใน parallel program โดยมีพื้นฐาน 3 แบบ

- SPMD : Single-Program-Multiple-Data เป็นแบบที่ component process จะมีการทำงานในตัวโปรแกรมที่เหมือนกัน แต่มีการใช้ data ที่มาจากคนละแหล่ง
- MPMD : Multiple-Program-Multiple-Data เป็นแบบที่ component process จะมีการทำงานตัวโปรแกรมคนละโปรแกรมกัน และมีการใช้ data ที่มาจากคนละแหล่งด้วย
- SIMD : ทั้ง SPMD และ MPMD program เป็น MIMD คือคำสั่งที่ต่างกันสามารถที่จะ execute ได้ใน process ที่ต่างกันในเวลาเดียวกัน ส่วน SIMD program เป็นแบบที่มีการ execute คำสั่งเดียวกันทั้งหมดของทุก process ในเวลาเดียวกัน

ส่วนใหญ่ในการเขียนโปรแกรมจะเป็นการทำงานแบบ MIMD program โดย MPMD program จะใช้การทำงานแบบที่เรียกว่า "parallel block" หรือการสร้าง multi-code ขึ้นมา ส่วนการโปรแกรมแบบ SPMD จะใช้การทำงานแบบที่เรียกว่า "parallel loop" หรือการสร้าง multi-data ขึ้นมา

มี 2 term ที่จะพบบ่อยๆคือ "data parallel" (การแตกข้อมูลเป็นส่วนย่อยๆ)จะใช้ในการทำงานแบบ SPMD program ส่วนอีก term หนึ่งคือ "functional-parallel" (การแตกการทำงานเป็นส่วนย่อยๆ)จะใช้ในการทำงานของ MPMD program ซึ่ง MPMD(functional-parallel) และ SPMD(data-parallel) จะสามารถใช้ร่วมกันใน parallel program ได้

Parallel Block จะถูกใช้ใน MPMD program เป็นการใช้ **parbegin** และ **parend** construct โครงสร้าง construct นี้ต้นแบบจะมีจากทฤษฎีของ Dijkstra และรู้จักกันใน *cobegin* และ *coend* โดยจะมีรูปแบบดังนี้

Parbegin S_1 S_2 ... S_n **parend**

โดยจะเรียกว่า parallel block เมื่อ S_1 S_2 ... S_n เป็น component process ที่มี code ที่แตกต่างกัน ซึ่งถ้าเป็น sequential block : begin S_1 S_2 ... S_n end

n component process ของ parallel block เริ่มการ execution จะเหมือนกับ parallel block ถูก execute ซึ่งการ execute ของแต่ละส่วนจะเป็นอิสระต่อกัน และอาจจะมีการติดต่อกันของ process ที่กำลัง execute อยู่ parallel block จะจบการทำงานเมื่อ ทุก component process เสร็จการทำงาน

Parallel Loop เมื่อทุก process ใน parallel block มีการใช้ code ที่เหมือนกันแล้ว เราสามารถที่จะนิยาม parallel block โดยการเขียนที่เรียกว่า parallel loop คือ

Parbegin Process(1)…Process(n) **parbegin**

ซึ่งสามารถที่จะเขียนใหม่ว่า

parfor (i := 1; i <= n; i++) { Process(i) }

parallel loop จะถูกใช้ใน SPMD parallel programs

process ใน OS หรือใน networking system จะเป็นแบบที่มีการ process มีการทำงาน code ที่ต่างกัน (heterogenneus) parallel program สามารถที่จะเป็นแบบที่ process ทำงาน code ที่เหมือนกัน หรือแบบที่ process ที่ทำงาน code ที่ต่างกันก็ได้ เราสามารถที่จะ ทำ MPMD ให้เป็น SPMD program ได้เช่น

```
parfor (i := 1; i <= n; i++) {
    if (i = 0) A;
    if (i = 1) B;
    if (i = 2) C;
}
```

Multi-Code versus Single-Code MPMD จะมีการใช้งานแบบที่ parbegin A;B;C; parend user ต้องมีการเขียนโปรแกรม 3 ครั้ง และ compile เพื่อที่จะให้ได้ 3 execute program A, B, C ซึ่งจะถูก load ไปให้แต่ละ node

```
Run A on node1
Run B on node2
Run C on node3
```

ซึ่งโปรแกรม A, B, C ก็เป็นโปรแกรมแบบ sequential บวกกับที่เรียก library ที่ใช้สำหรับการติดต่อกัน

SPMD program จะอยู่ในลักษณะ single-code คือ **parfor** (i := 1; i <= N; i++) { foo(i)}

User จะเขียนโปรแกรมเพียงครั้งเดียว เช่น

```
Pid = my_process_id();
Numproc = number_of_processes();
for (i := 1; i <= N; i+numproc) { foo(i)}
```

โปรแกรมนี้จะถูก compile เพียงครั้งเดียว และจะถูก load ไปยัง n node ทำให้สามารถที่จะ execute program เดียวกันได้ใน node ที่ต่างกัน

6.2 Static versus Dynamic Parallelism

โครงสร้างของโปรแกรมจะเกี่ยวข้องกับส่วนประกอบของตัวโปรแกรมเอง เช่น code **if** (C) S1 **else** S2 จะมีโครงสร้างของคำสั่ง "**if** (…) … **else** …" และมีส่วนประกอบสามอย่างคือ C, S1, S2 โปรแกรมจะแสดงลักษณะของ static parallelism ถ้าโครงสร้างของมันและจำนวนส่วนประกอบของมัน สามารถที่จะพิจารณาได้ก่อนที่จะมีการ run แต่ถ้าไม่สามารถที่จะพิจารณาได้ โปรแกรมนั้นจะมีลักษณะการทำงานที่แบบ dynamic parallelism

static parallel program จะใช้เวลาในการ run และใช้การทำงานที่น้อยกว่า dynamic parallel program เพราะไม่ว่าจะเป็น การจัดแบ่ง memory, การจัดแบ่ง stack, system table สามารถที่จะทำได้ก่อนที่นะ run แต่ใน dynamic ในส่วนการจัดแบ่งต่างๆ จะต้องให้มีความยืดหยุ่นเพียงพอที่จะสามารถ run โปรแกรมได้

static parallelism สามารถที่จะแสดงได้ดัง 6.1 (parallel block, parallel loop) ส่วน dynamic parallelism จะเป็นชนิดของ fork และ join operation

Fork/Join : ตัวอย่างที่จะทำให้มอง operation นี้ได้คือ ถ้าโปรแกรมมี 3 process โดย A เป็น main process และจะถูกสร้างโดยอัตโนมัติเมื่อมีการ execute program

```
Process A;              Process B;              Process C;
begin                   begin                   begin
  Z := 1;               fork(C)                   Y := boo(Z);
  fork(B);                X := foo(Z);          end
  T := foo(3);            join(C);
end                       output(X+Y);
                        end
```

รูปที่ 6.1 ตัวอย่างการทำงานของ fork และ foo

จากรูปที่ 6.1 process A execute คำสั่ง **fork**(B) จะมีการสร้าง process B ขึ้น โดยจะทำงานไปควบคู่กับ A ก็คือ ขณะที่ B กำลัง execute A สามารถที่จะทำการ execute คำสั่งถัดไปได้เลย (T := foo (3)) เราจะเรียก A ว่าเป็น parent process ของ B และ B จะเป็น child process ของ A 2 process จะมีการ execute แบบ asynchronously คือ A สามารถที่จะจบการทำงานก่อน B ได้ แต่โดยทั่วไปแล้ว parent process ไม่สามารถที่จะจบการทำงานก่อน child process ได้

บางครั้งเราต้องการให้ parent process รอ child ก่อนที่ parent จะจบการทำงานได้ เราสามารถที่จะใช้ **join** statement ใน code ตัวอย่าง process B จะมีการสร้าง process C ขึ้นมาอีก และมี **join** statement

ถูกใส่เข้าไปก่อน output statement ใน code ของ process B ซึ่ง **join** statement จะทำให้ B จะต้องรอให้ C จบการทำงานก่อน ถึงจะไปทำงานในคำสั่งถัดไป

Fork และ Join มีความยืดหยุ่นในการทำงาน แต่อย่างไรก็ดีคำสั่งเหล่านี้ ไม่มีลักษณะความเป็นโครงสร้าง (unstructured) ใน sequential program ถ้าจะมีการใช้ 2 statement นี้อย่างระมัดระวัง ส่วนใน parallel program จะมีการใช้ในส่วนของ parallel block และ parallel loop เท่านั้น

6.3 Process Grouping

Process ใน parallel program บ่อยครั้งที่จะไม่เป็นอิสระ คือมันจะต้องมีการติดต่อสื่อสารกับ process อื่น

Process group เป็น กลุ่มของ process และจำนวนของ member ใน group จะเป็นขนาดของ group และแต่ละ group จะมีหมายเลขของแต่ละgroup(group ID) ซึ่งจะไม่มีการซ้ำการระหว่าง process ใน parallel program แต่ละ process ใน group จะมีลำดับใน group (rank) โดยจะเป็น integer เริ่มตั้งแต่ 0 ถึง n-1 เมื่อ n เป็น ขนาดของ group

ใน parallel program จะมี function ในการสร้างและทำลาย group และมี function ในการค้นหา group ID, group membership, member rank

6.4 Allocation Issue

parallel program จะมีการคำนวณ(workload) บนข้อมูล การ Allocation จะเกี่ยวข้องกับการแบ่ง data และ workload ให้แก่แต่ละ process และการ map process ให้กับแต่ละ node(processors)
Parallelism เป็นงานส่วนที่สำคัญในการแบ่งโดยจะเลือกคุณสมบัติของ parallelism และ granularity Degree ของ parallelism (DOP) ของ parallel program จะนิยามโดยจำนวน component process ที่สามารถที่จะ execute ไปด้วยกันได้
Granularity (Grain size)จะนิยามโดยการคำนวณของ workload ระหว่าง 2 parallelism หรือ 2 กระบวนการที่ติดต่อกัน

หน่วยของ granularity เป็นจำนวนคำสั่ง, จำนวนกระบวนการของ floating-point โดย grain size จะถูกเรียกได้ คือ fine(small), medium และ coarse(large) โดยถ้า grain size มีขนาดน้อยกว่า 200, medium จะอยู่ระหว่าง 200-2000, และ large จะตั้งแต่ 2000 ขึ้นไป

granularity บางครั้งจะเรียกว่า process size, จำนวนรวมของการทำงานของ component process
Implicit and Explicit Allocation ถ้าเป็น explicit allocation นั้น user จะเป็นผู้จัดแบ่ง data และ workload เอง แต่ถ้าเป็น implicit allocation งานในส่วนนี้จะขึ้นอยู่กับ Compiler และ runtime-system ซึ่งอาจมีการใช้งานรวมกันก็ได้

บทที่ 7
ทฤษฎีของการติดต่อสื่อสาร

7.1 Interaction Operation

จะมีกระบวนการ interaction ที่เกิดขึ้นบ่อยอยู่ สามแบบ

communication : กระบวนการ communication จะส่งผ่าน data ระหว่าง 2 หรือ 3 process ใน shared-memory system program process หนึ่งสามารถที่คำนวณค่าตัวแปร และเก็บไว้ใน shared memory ถ้า process อื่นต้องการเอา data นี้มาจะเรียกการ communication นี้ว่า "shared variable"

ใน multiprocessor program จะอนุญาตให้ parent process สามารถที่จะสร้าง child process โดยการเรียกใช้ function fork() ในการ execute fork() data สามารถที่จะส่งผ่านระหว่าง child process และ parent จะเรียกการ communication นี้ว่า "parameter passing" และท้ายสุดใน multicomputer model process จะมีการ communication ระหว่าง process ซึ่งจะเรียกว่า "message passing"

synchronization : กระบวนการ synchronization จะทำให้ process รอ process อื่น หรือ อนุญาตให้ รอที่จะ execute ได้ โดยจะมีลักษณะที่แตกต่างกันของการ synchronization อยู่ 3 แบบ

- Atomicity : process จะต้องทำกระบวนการตามลำดับเหมือน *กระบวนการ atomic*(atomic operation) คือ เปรียบเสมือนกระบวนการเดียวตามดังนี้

```
parfor (i := 1; i <= n; i++) {
        atomic { x = x+1; y = y-1; }
}
```

- Control Synchronization process execute กระบวนการ control synchronization จะรอจนกระทั่ง program execute ถึง *control states* ตัวอย่างง่ายที่จะแสดงให้เห็นคือ *barrier synchronization*

```
parfor (i := 1; i <= n; i++) {
        P_i
        barrier
        Q_i
}
```

ถ้าเรามี n processor process *i*th process จะ execute P*i* ตามด้วยคำสั่ง **barrier** และตามด้วย Q_i เมื่อ P_i จบการทำงานแล้วจากนั้นเมื่อมาเจอคำสั่ง **barrier** มันจะมีการรอจนกระทั่ง process อื่นๆ ทำงานถึง **barrier** ด้วย

- Data Synchronization สำหรับ process จะ execute กระบวนการ data synchronization และจะรอจนกระทั่ง program execute จนถึง data states ตาม data synchronization นั้น เช่น process execute wait(x > 0) statement จะรอจนกว่าตัวแปร x มีค่าเป็นบวก

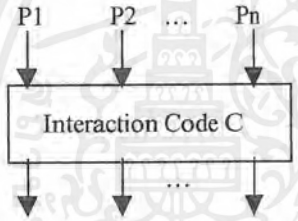
Aggregation : เป็นกระบวนการที่ใช้รวมผลลัพธ์ที่ได้จากแต่ละ component process ของ parallel program เพื่อที่จะสร้างผลลัพธ์ที่สมบูรณ์ ตัวอย่างเช่น

```
parfor ( i := 0; i <= n; i++) {
        x[i] := A[i] * B[i];
        inner_product := aggregate_sum(x[i]);
}
```

โดย **aggregate_sum** จะรวมผลลัพธ์ที่ของ x[0],x[1],…,x[n-1] เพื่อเป็นผลลัพธ์สุดท้าย

7.2 Interaction Modes

เมื่อมี n process ที่ interact กัน โดย execute interaction code C โดย process จะเรียกว่า *parties* or *participants* ของ interaction และเราจะเรียกการ interaction แบบ synchronous ถ้า code C ไม่สามารถที่จะ execute ได้จนกว่า participants ทุกตัว ทำงานจนถึง C และเราจะเรียกการ interaction แบบ asynchronous ถ้าเมื่อ process ทำงานจนถึง C แล้ว มันสามารถที่จะทำงานต่อไปได้เลย โดยไม่ต้องรอ process อื่น



รูปที่ 7.1 Interaction Modes

Parallel program ส่วนมากจากทำงานแบบ synchronous interaction เพราะง่ายที่จะเข้าใจเพราะเมื่อ process execute C จะรู้ได้เลยว่า ทุก process ทำงานมาถึง C แล้ว แต่สำหรับ asynchronous interaction จะมีความยืดหยุ่นในการทำงานมากกว่า

บทที่ 8
อัลกอริทึมที่ใช้ศึกษา

8.1 Algorithm ที่ใช้ในโปรแกรมในการคำนวณหาค่า Pi

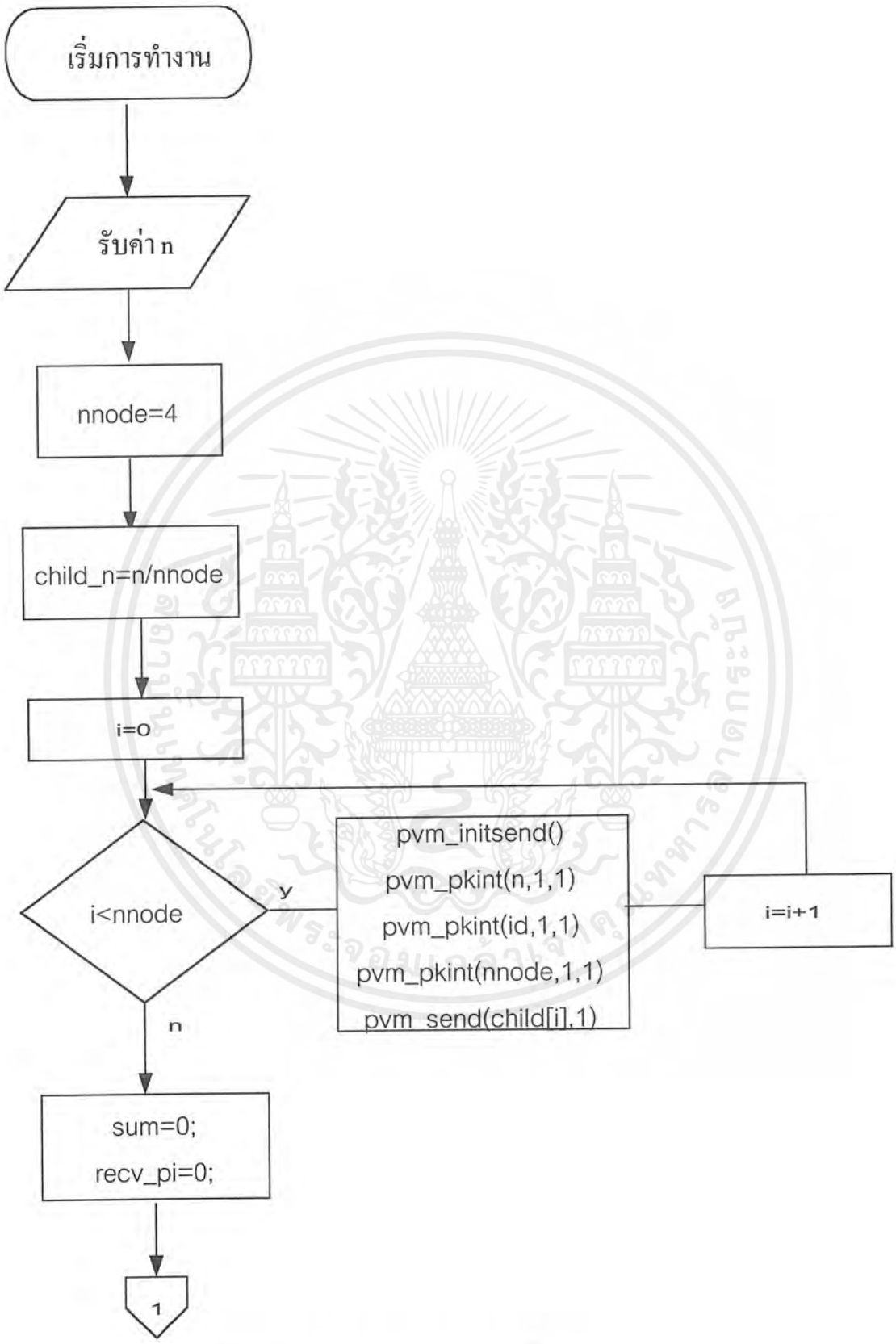
เป็น Algorithm ที่ใช้ในการคำนวณหาค่า Pi โดยค่าของ Pi จะหาได้จากการหาค่าของสมการ

$$f(x) = \int (4 / (1+ x^2)) \qquad (1)$$

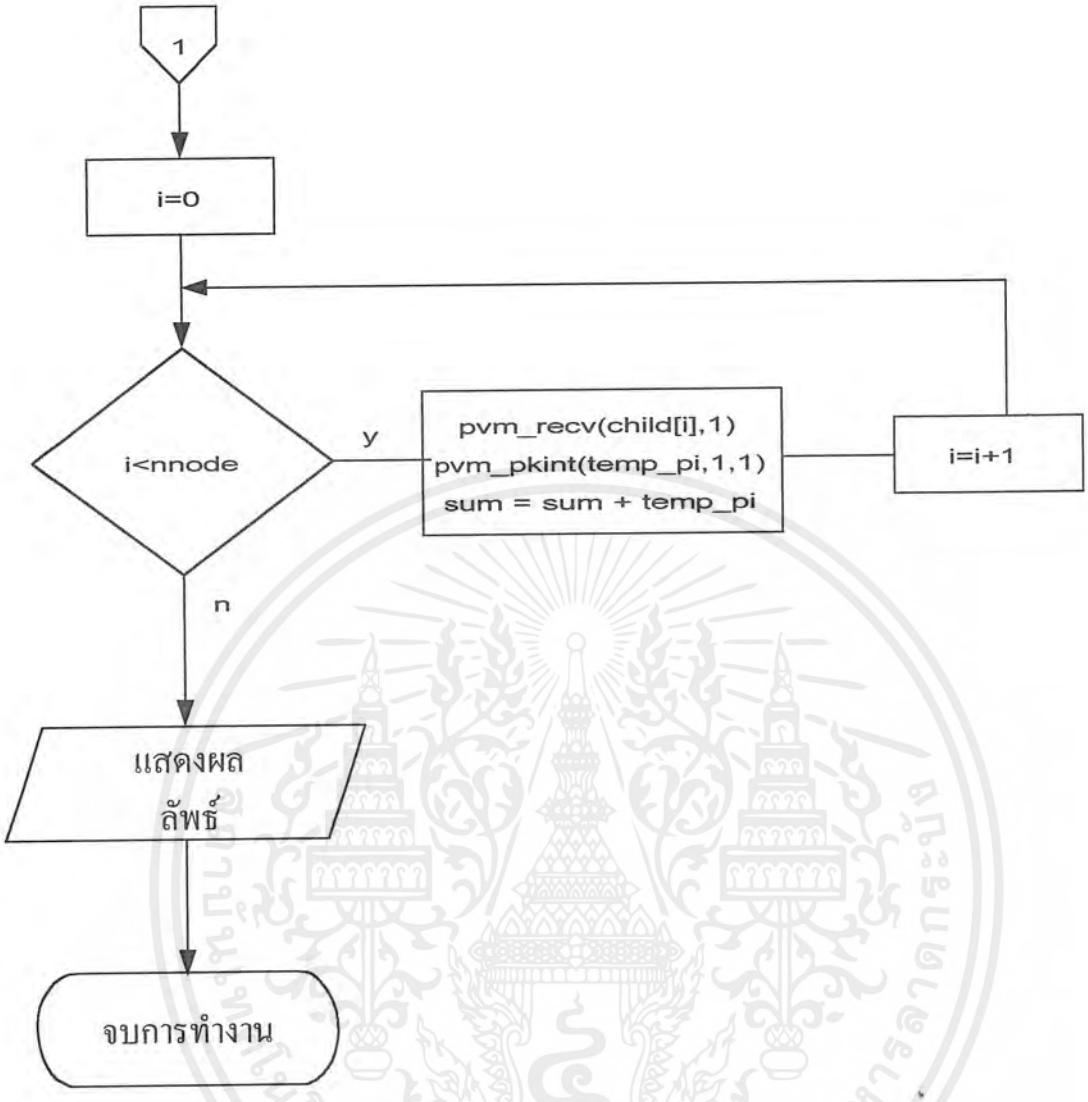
โดยทำการอินทีกรัลตั้งแต่ช่วง 0 ถึง n โดย n ยิ่งมีค่ามากค่าของ Pi ที่ได้ยิ่งมีค่าที่ใกล้ค่าจริงมากที่สุด

ในโปรแกรมจะให้ป้อนค่า n ที่ต้องการให้อินทิกรัลจาก 0 ถึง n จากนั้น เครื่องพ่อ (Master) ก็ทำการแบ่งช่วงจาก 0 ถึง n เป็นช่วงๆ ตามจำนวนเครื่องลูก (Child) ที่จะให้ร่วมกันช่วยประมวลผล จากนั้นก็ช่วงแต่ละช่วงไปให้เครื่องลูก (Child) แต่ละเครื่อง แล้วให้เครื่องลูก (Child) แต่ละเครื่องประมวลผลในช่วงของตัวเองที่ได้รับ จากนั้นก็ส่งผลลัพธ์ที่ได้จากการคำนวณในช่วงของเครื่องแต่ละเครื่อง กลับไปให้ เครื่องที่เป็นเครื่องพ่อ (Master) รับผลลัพธ์แล้วรวมผลลัพธ์เป็นผลลัพธ์สุดท้าย

ขั้นตอนการทำงานของเครื่องพ่อ (Master)

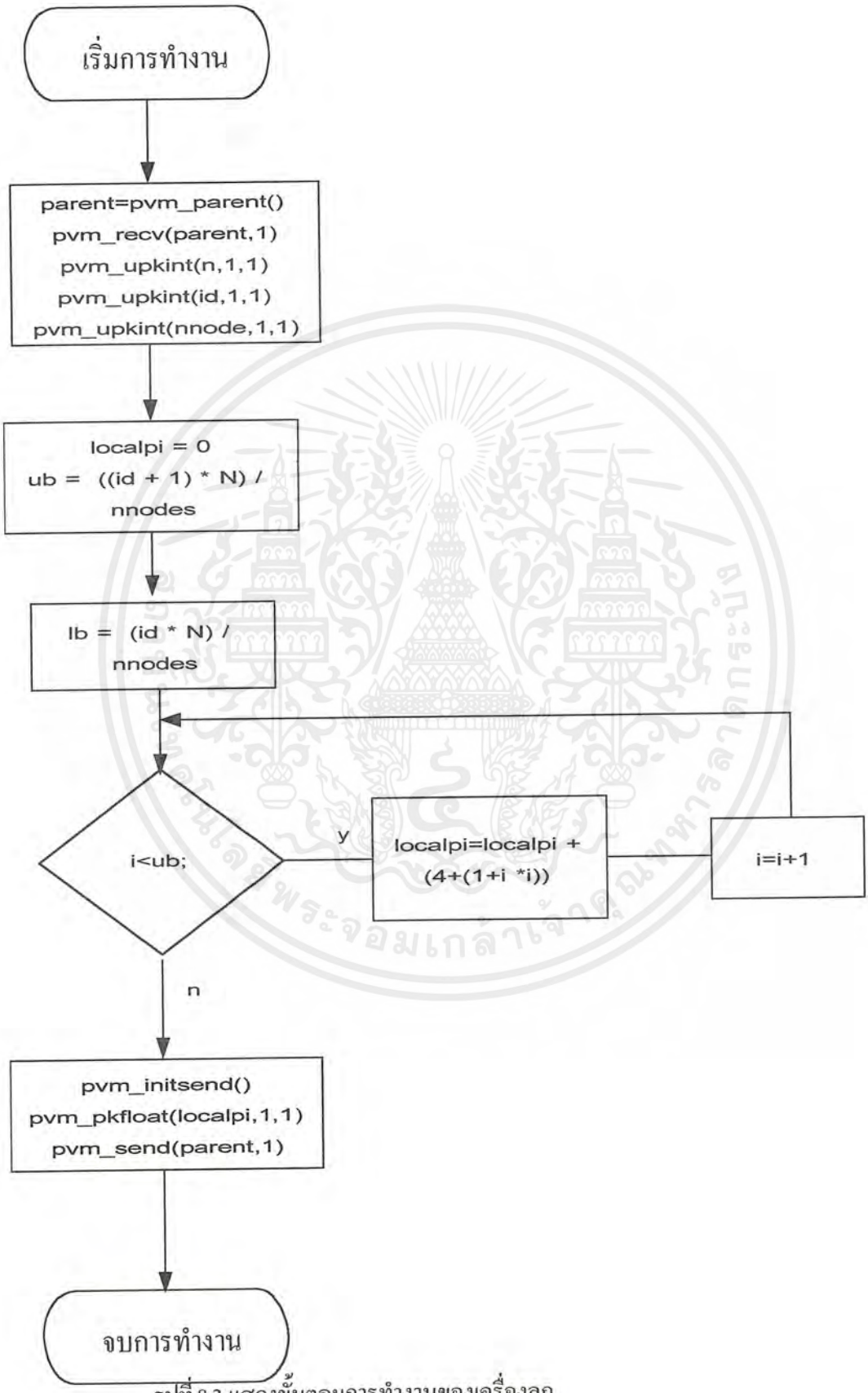


รูปที่ 8.1 แสดงขั้นตอนการทำงานของเครื่องพ่อ



รูปที่ 8.2 แสดงขั้นตอนการทำงานของเครื่องพ่อ(ต่อ)

ขั้นตอนการทำงานของเครื่องลูก (Child)

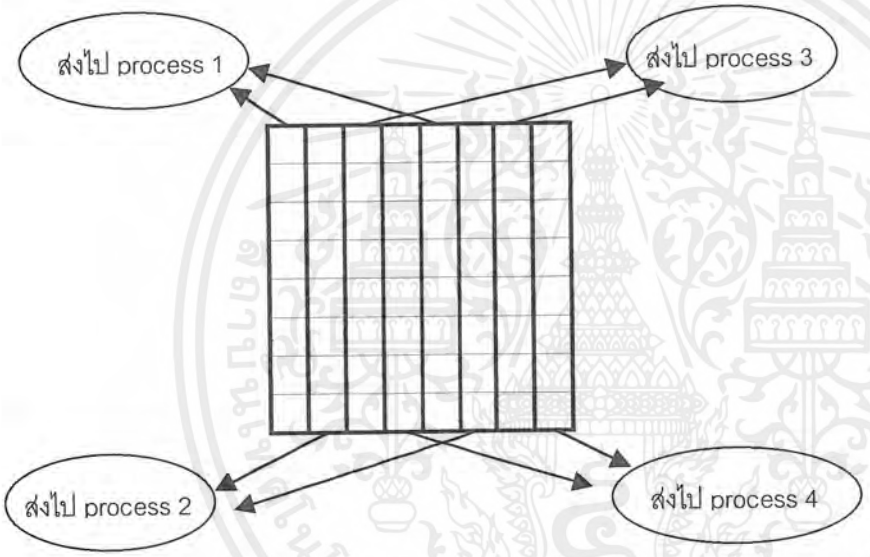


รูปที่ 8.3 แสดงขั้นตอนการทำงานของเครื่องลูก

8.2 Algorithm ที่ใช้ในการคำนวณ Metrix x Vector

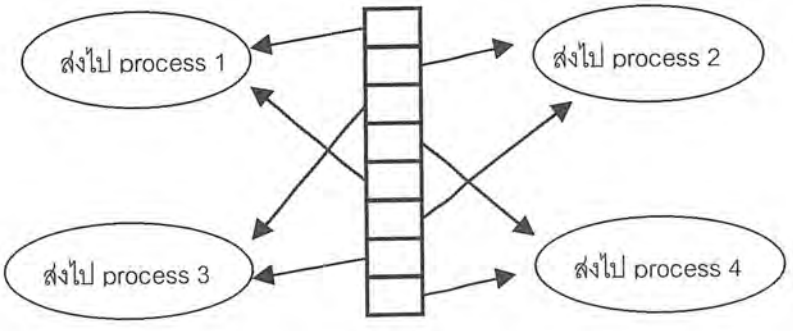
Algorithm นี้จะประกอบด้วยโพรเซสเริ่มต้น (master process) และ โพรเซสลูก (child process) ซึ่งที่เครื่องพ่อ (master) จะมีการสร้างโพรเซสอื่นๆขึ้นมาจากนั้นโพรเซสแรกจะทำการรับจำนวนมิติของเมทริกซ์ จากนั้นก็ส่งมิตินี้ไปให้โพรเซสลูกทั้งหมด จากนั้นโพรเซสแรกจากทำการกำหนดให้เมทริกซ์ A มีค่า 3 ทุกค่า และทำการกำหนดเวคเตอร์ B มีค่า 3 ทุกค่า จากนั้นก็ทำการแบ่งเมทริกซ์ A เป็นส่วนและส่งไปให้โพรเซสอื่นๆ ที่รับผิดชอบแต่ละส่วน โพรเซสแต่ละโพรเซสก็จะรับแล้วนำมาคำนวณในส่วนที่ตัวเองได้รับ จากนั้นในโพรเซสแรกก็จะทำการรับเวคเตอร์ C ในส่วนต่างๆที่ส่งไปให้โพรเซสอื่นทำ แล้วก็ทำก็รวมเวตเคอร์ C ให้สมบูรณ์จากนั้นก็เขียนลงไฟล์ vec_metrix_c

* หลักการแบ่งเมทริกซ์ A ถ้ามิติเป็น 8 x 8 และให้มีโพรเซสทั้งหมด 4 โพรเซส



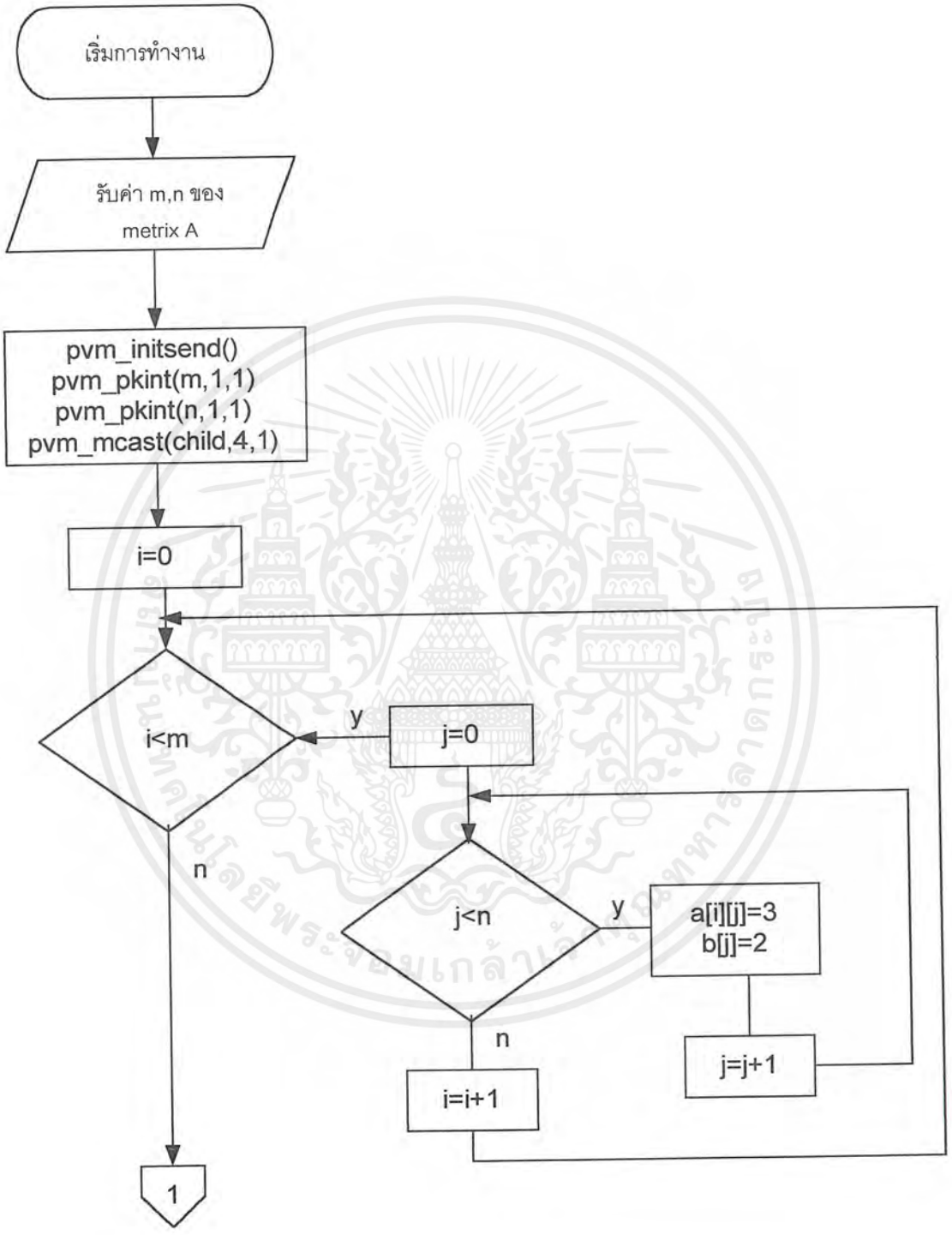
รูปที่ 8.4 แสดงการแบ่งแต่ละ column ของเมทริกซ์ A

* หลักการแบ่งเวคเตอร์ B ให้มีโพรเซสทั้งหมด 4 โพรเซส

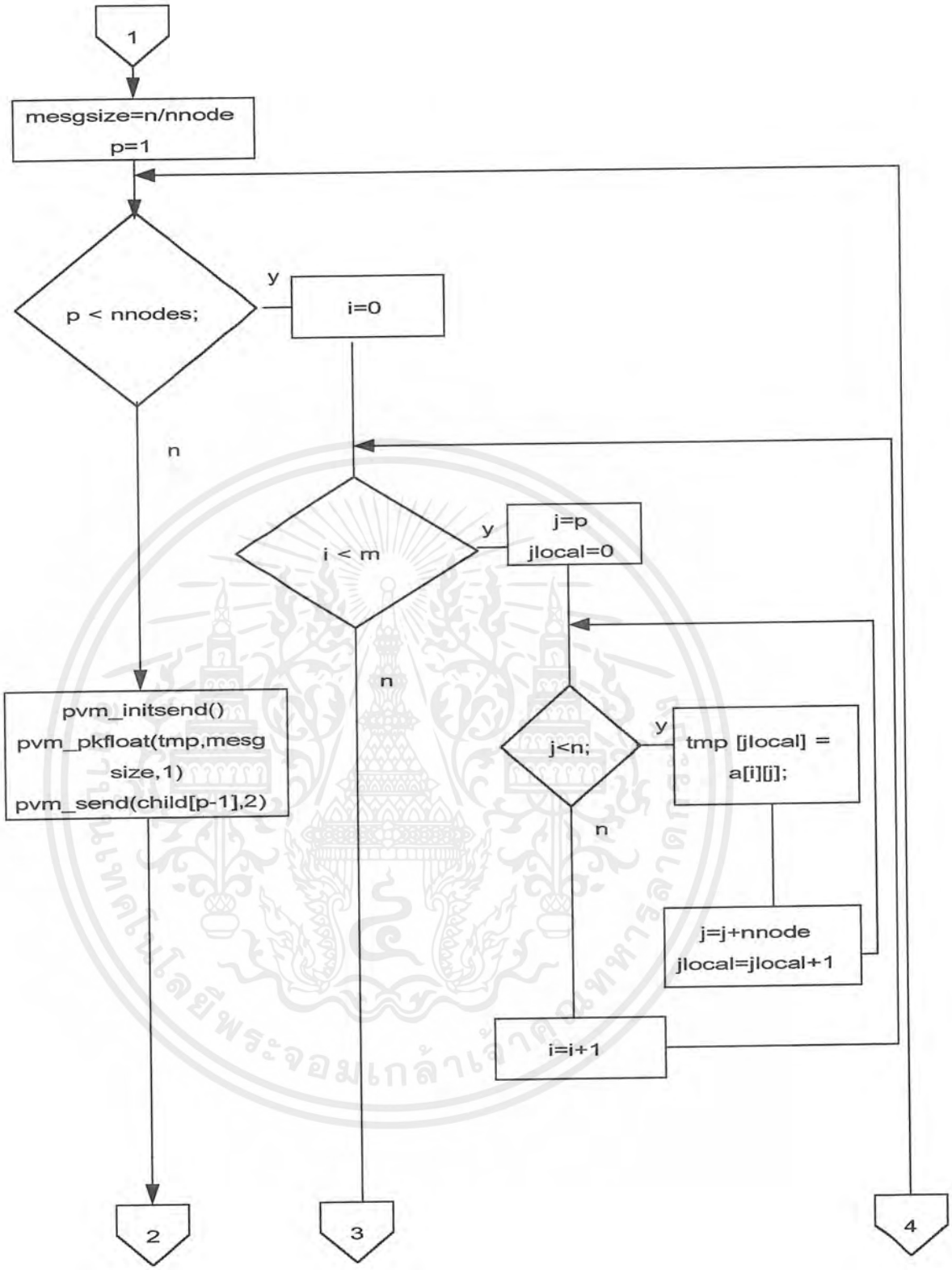


รูปที่ 8.5 แสดงการแบ่งแถวของเวคเตอร์ B

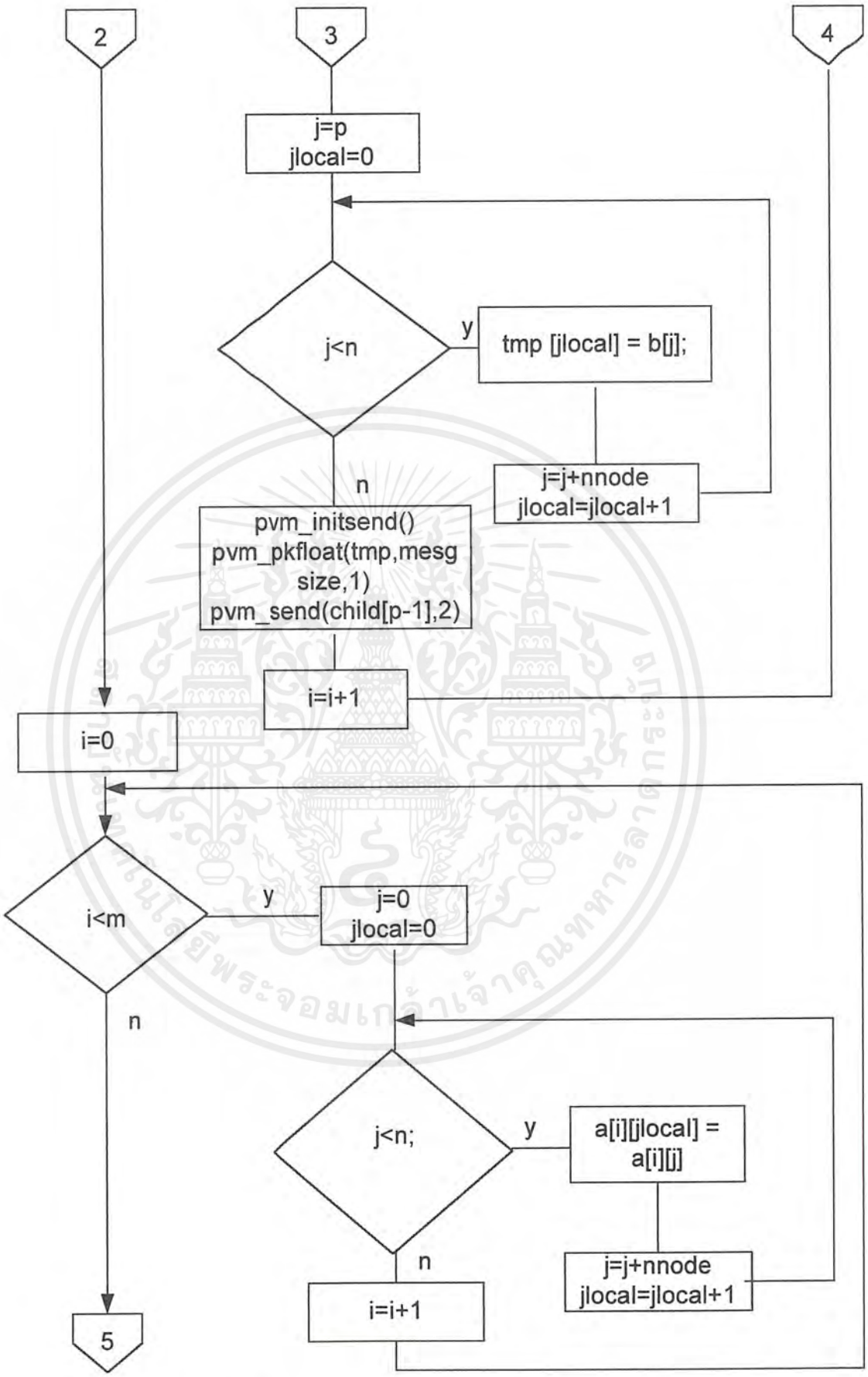
ขั้นตอนการทำงานของโพรเซสแรก



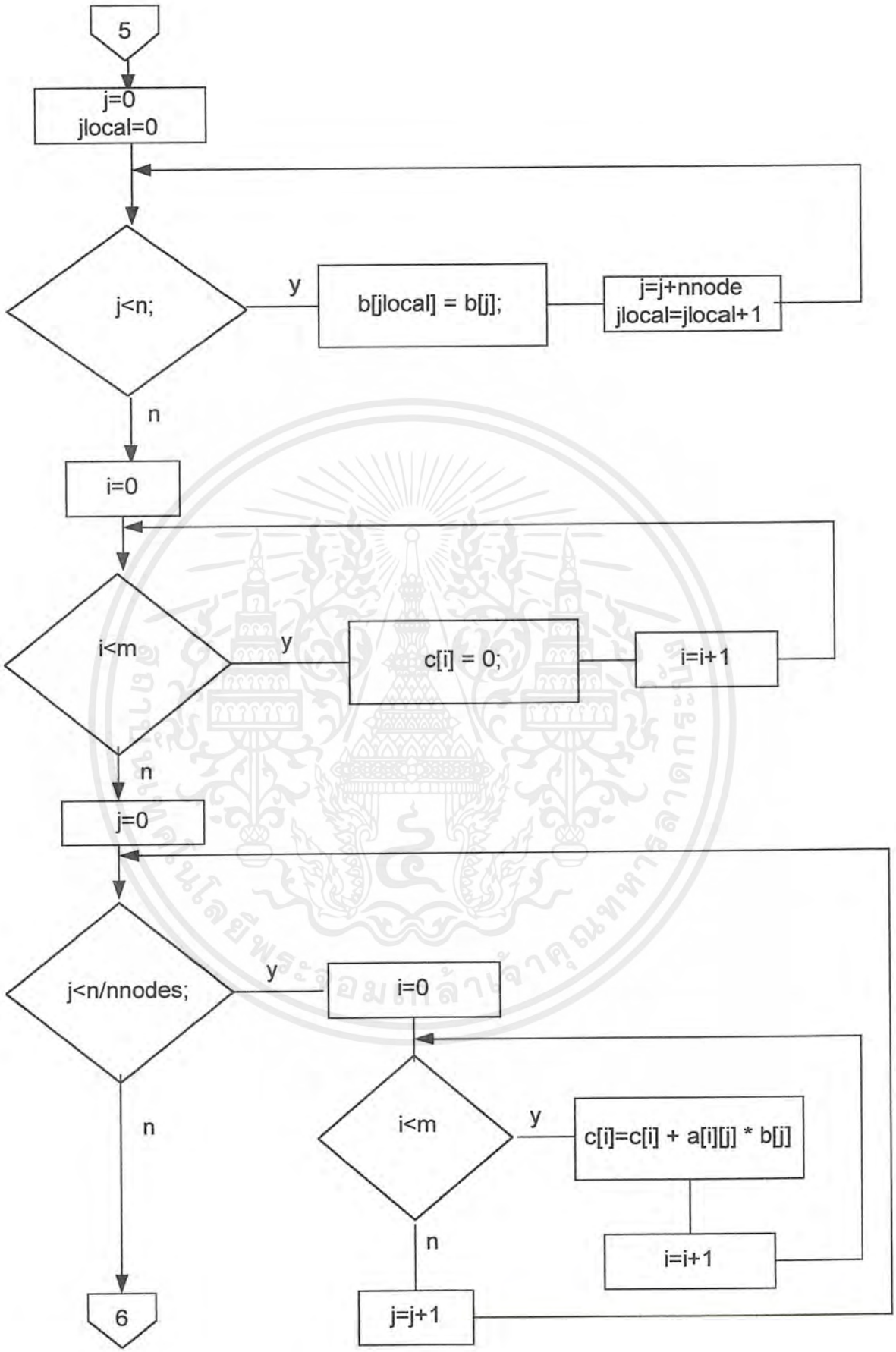
รูปที่ 8.6 แสดงขั้นตอนการทำงานของโพรเซสแรก



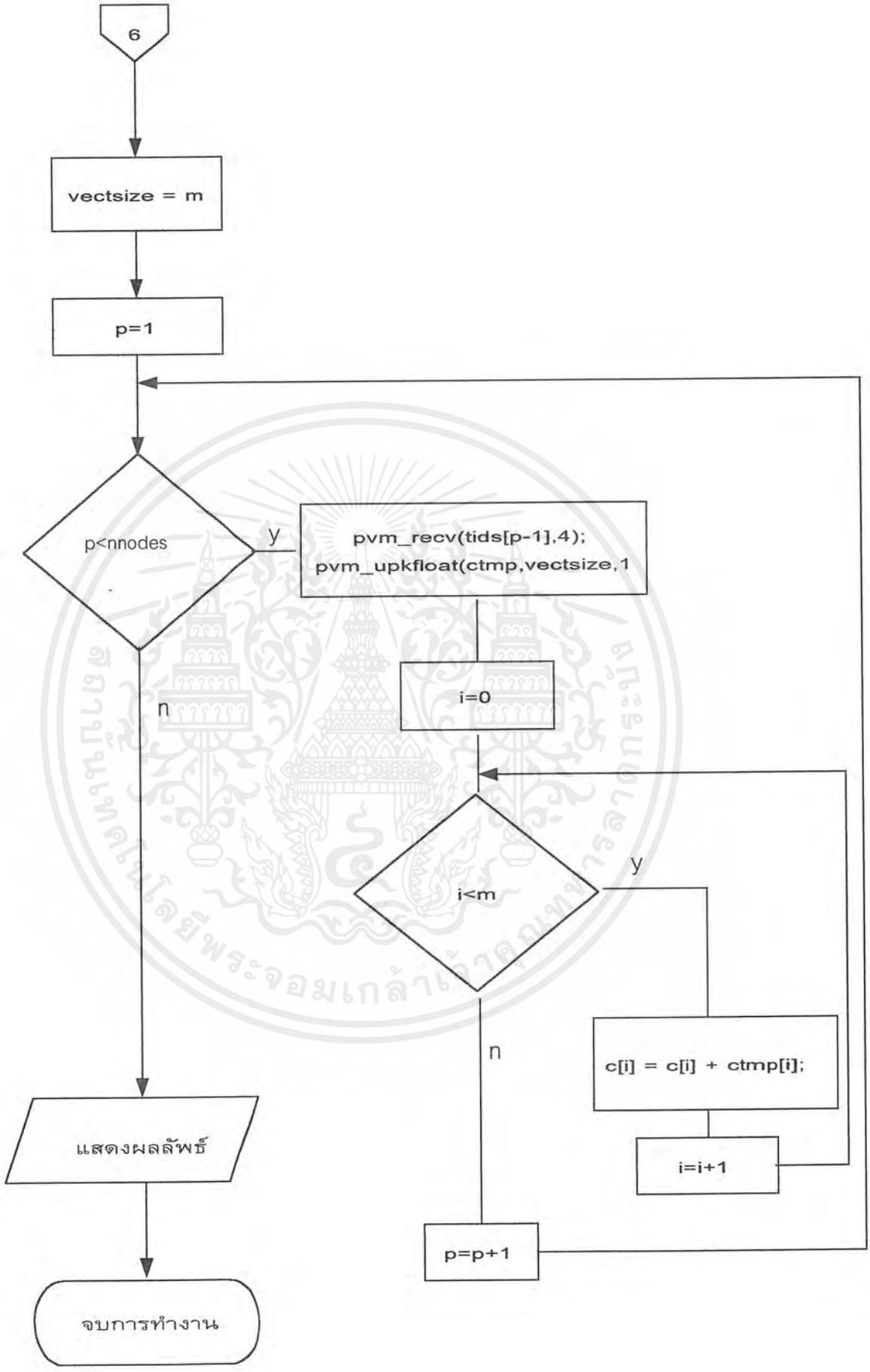
รูปที่ 8.7 แสดงขั้นตอนการทำงานของโพรเซสแรก(ต่อ)



รูปที่ 8.8 แสดงขั้นตอนการทำงานของโพรเซสแรก(ต่อ)

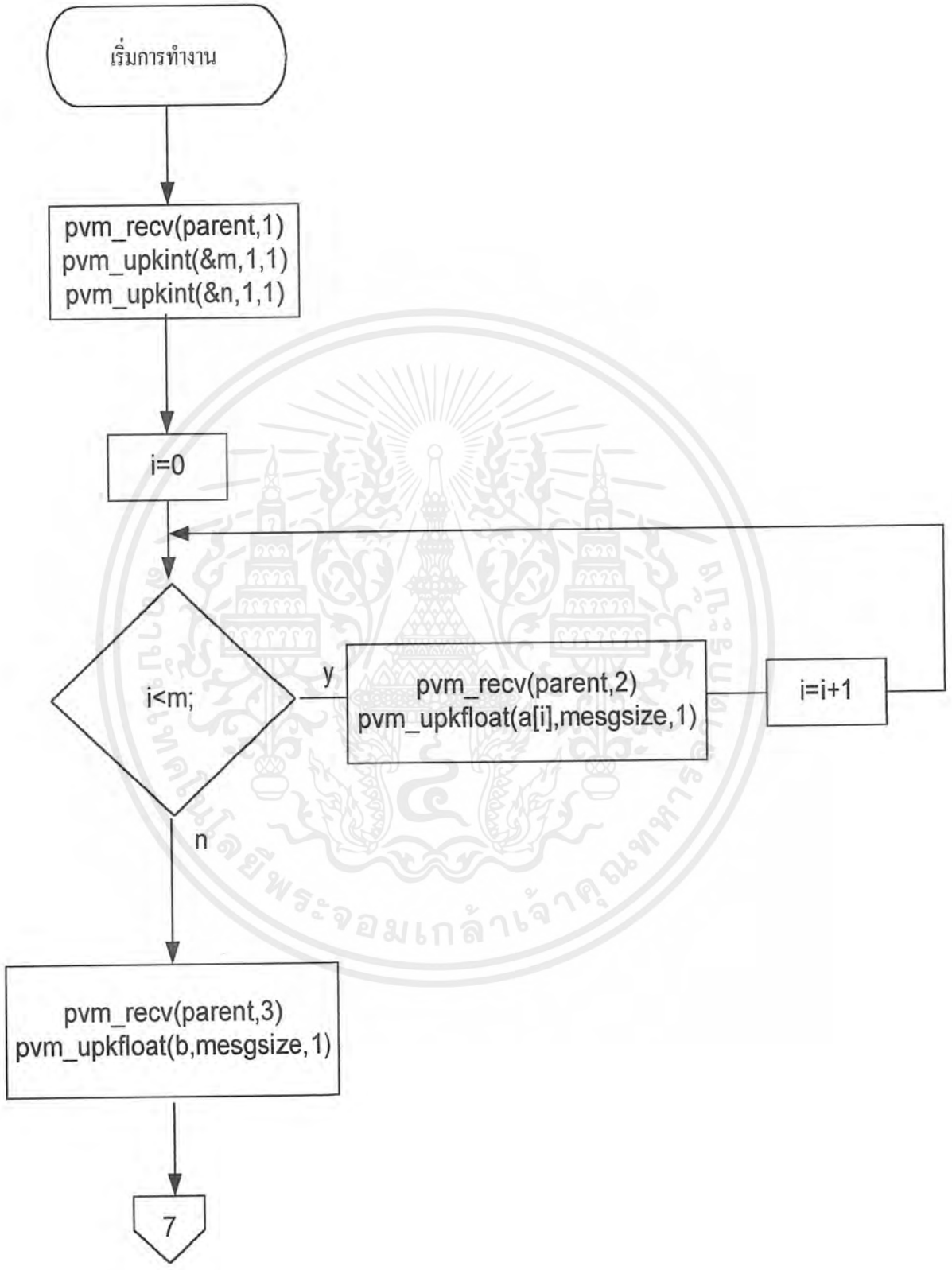


รูปที่ 8.9 แสดงขั้นตอนการทำงานของโพรเซสแรก(ต่อ)

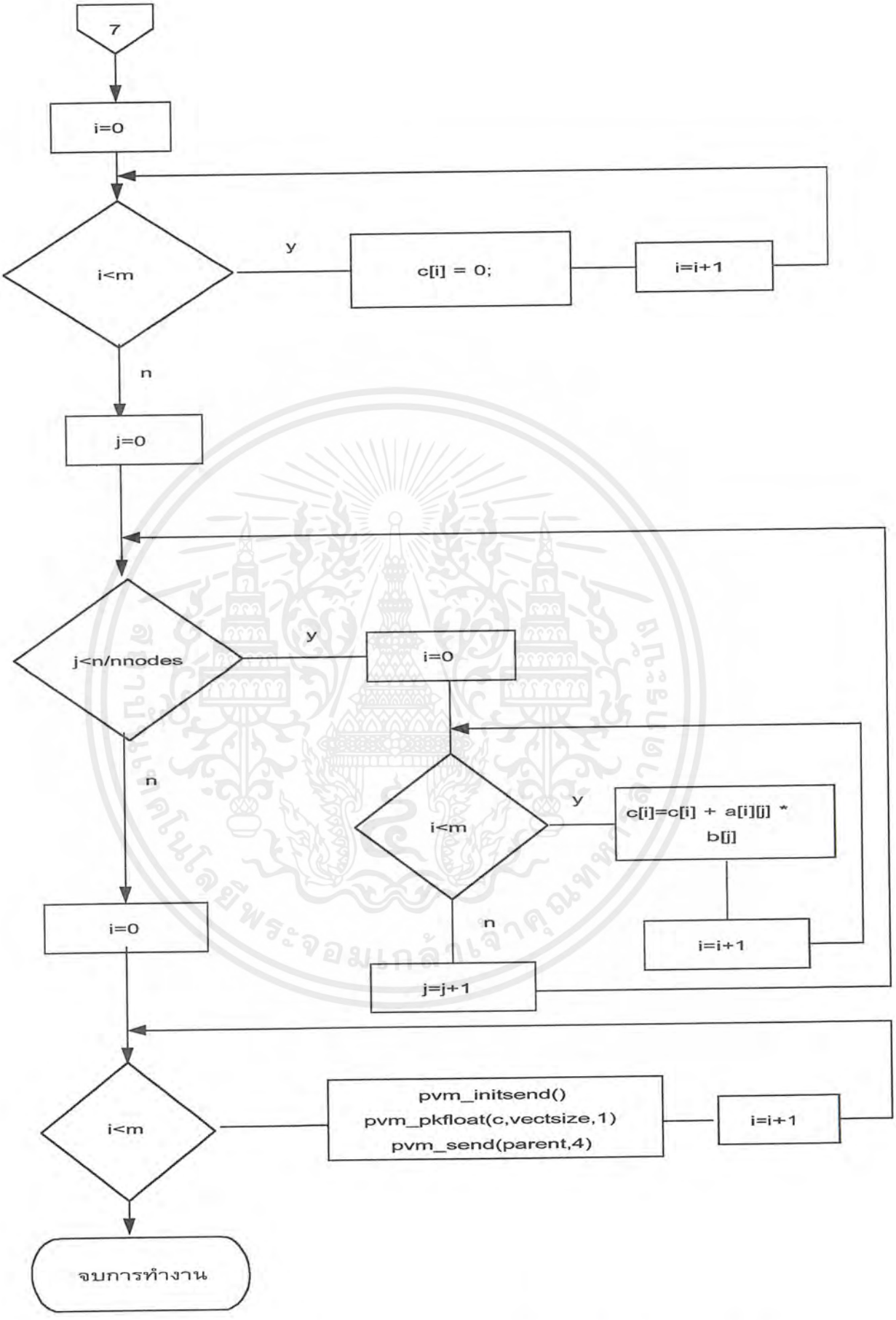


รูปที่ 8.10 แสดงขั้นตอนการทำงานของโพรเซสแรก(ต่อ)

ขั้นตอนการทำงานของโพรเซสอื่นๆที่เหลือ



รูปที่ 8.11 แสดงขั้นตอนการทำงานของโพรเซสอื่นๆ ที่ไม่ใช่โพรเซสแรก



รูปที่ 8.12 แสดงขั้นตอนการทำงานของโพรเซสอื่นๆ ที่ไม่ใช่โพรเซสแรก (ต่อ)

บทที่ 9
รูปแบบและวิธีการทดลอง

9.1 Environment

9.1.1 โดยศึกษาหาข้อมูลต่างๆจาก web site ต่างๆตาม internet และ หนังสือการโปรแกรมบนระบบปฏิบัติการ UNIX ที่มีวิธีการเขียนโปรแกรมแบบขนาน

9.1.2 ข้อมูลต่างๆที่ได้จะเป็นรูปแบบเอกสาร โดยได้นำข้อมูลที่ได้มาทดลองใช้ดูว่าข้อมูลไหนสามารถนำไปใช้งานได้ และ แบบไหนใช้ได้ง่ายกว่ากัน

9.1.3 เครื่องมือที่ใช้ ได้แก่

Architecture	Personal computer Pentium II 400 Mhz จำนวน 4 เครื่อง , Ethernet Hub 10 M จำนวน 1 ตัว, Ethernet Switch10/100 M จำนวน 1 ตัว, สาย UTP Cat5 จำนวน 4 เส้น, LAN card 10/100 จำนวน 4 card
Operating System	ใช้ระบบปฏิบัติการ LINUX ทุกเครื่อง และต้องมี C compiler ซึ่งการโปรแกรมแบบขนาน นี้ใช้ C compiler
Library	ต้องติดตั้ง library เพิ่มเติมเพื่อให้ C compiler สามารถมีการทำงานแบบขนานได้ โดย Library นี้ใช้ในการติดต่อกันระหว่างเครื่องและการโปรแกรมแบบขนาน และยังใช้ในการควบคุมการแบ่งงาน และควบคุมการส่งเมสเสจระหว่างเครื่อง โดยมี library ที่เป็นที่นิยมและให้เลือกใช้อยู่ 2 library คือ PVM และ MPI ซึ่งในที่นี้เลือกใช้ PVM เนื่องจากสามารถ run ได้หลาย operating system ได้คือ สามารถนำเครื่องที่มี operating system ต่างกันมาเชื่อมต่อกันได้
Algorithm	เป็น algorithm แบบขนานโดยเลือกใช้ algorithm คำนวณหาค่า Pi ที่ทำงานแบบขนาน , Algorithm ที่ใช้คำนวณเวคเตอร์คูณเมทริกซ์ , เพื่อใช้ในการทดสอบประสิทธิภาพกับโปรแกรมแบบขนานและแบบลำดับ
Program	โปรแกรมที่ใช้ทดสอบ forkjoin, Pi, Vector x Metrix

9.2 วิธีการ

9.2.1 โดยเริ่มจากการเชื่อมต่อ Personal computer แต่ละเครื่องเข้าด้วยกันโดยเชื่อมต่อกัน เป็นวง LAN เดียวกันโดยทดลองใช้ทั้งกับ Hub และ Switch เนื่องจากทั้งสองอย่างมีความเร็วในการรับส่งข้อมูลไม่เท่ากันซึ่งจะทำให้มีผลกับเวลาในการประมวลผลหรือไม่

9.2.2 นำ library PVM มาติดตั้งลงในเครื่อง computer ทุกเครื่อง หรือสามารถใช้ file INSTALL1

9.2.3 ทดสอบการspwan task ระหว่างเครื่องว่าได้หรือไม่โดยใช้โปรแกรมทดสอบ forkjoin

9.2.4 ศึกษาคำสั่งของ library PVM ต่างๆที่ใช้ในการเขียนโปรแกรมแบบขนาน

9.2.5 ศึกษา algorithm ของโปรแกรมแบบขนาน

9.2.6 เขียนโปรแกรมแบบขนานโดยเริ่มจากเขียนโปรแกรม Pi และ Vector x Metrix ตามลำดับ

9.2.7 ทดสอบเปรียบเทียบประสิทธิภาพระหว่างโปรแกรมแบบลำดับกับแบบขนานโดยใช้เวลาในการประมวลผลเป็นตัวเปรียบเทียบ

9.2.8 ทดสอบว่าความเร็วในวง network มีผลต่อเวลาในการประมวลผลงานแบบขนานโดยใช้ Hub และ Switch ในการทดสอบ

9.2.9 การศึกษาจะแบ่งการศึกษาออกเป็น 2 ส่วนคือ การศึกษาบนระบบเครือข่ายที่มีความเร็ว 10 M และ การศึกษาบนระบบเครือข่ายที่มีความเร็ว 100 M

9.2.10 การจับเวลาจะใช้ฟังก์ชั่น Time() ของ C Compiler บน LINUX โดยนำค่าเวลาเมื่อเริ่มตั้งแต่สร้าง process สำหรับทำการคำนวณก็จะเริ่มจับเวลาทันทีและนำค่าเวลาเมื่อเสร็จ จากนั้นมาหักลบการกันก็จะได้เวลาที่ทำการคำนวณ

9.2.11 ค่าเวลาที่ได้จะมาจากการ run โปรแกรมต่าง 5 ครั้งแล้วหาค่าเฉลี่ยเวลาทั้งหมดจาก 5 ครั้ง

9.3 ผลการทดลอง

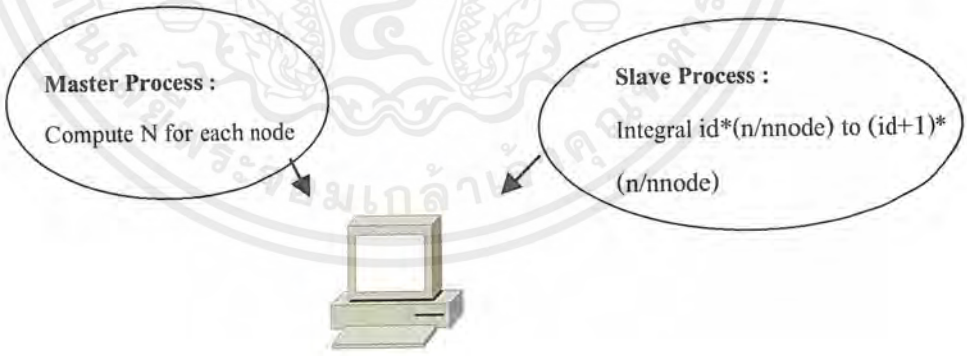
9.3.1 โปรแกรมคำนวณหาค่า Pi

9.3.1.1 ประมวลผลโปรแกรมที่ทำงานแบบตามลำดับ

จำนวนเครื่อง	เวลาที่ใช้(วินาที)	หมายเหตุ
1	20	Sequential

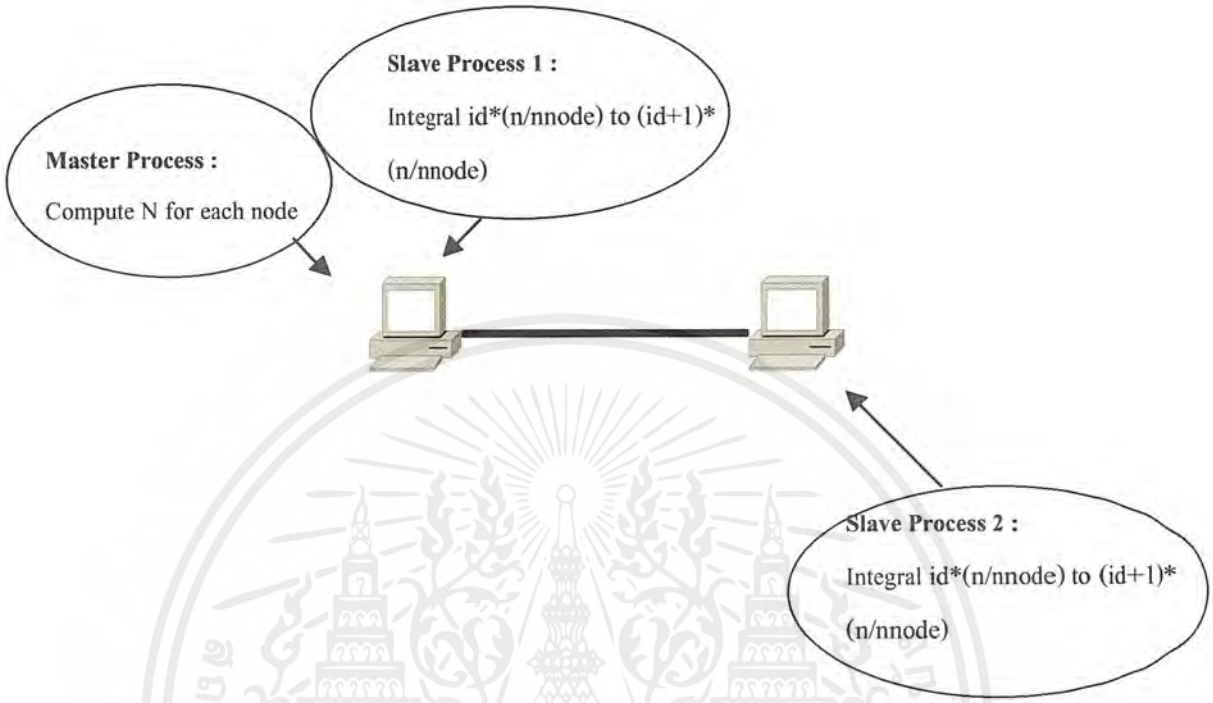
ค่า PI ที่ได้ = 3.14159267359042670000

9.3.1.2 โมเดลของการประมวลผลโปรแกรมที่ทำงานแบบขนานโดยใช้เครื่องพิวเตอร์ 1 เครื่องที่มีโพรเซสเซอร์เดียว



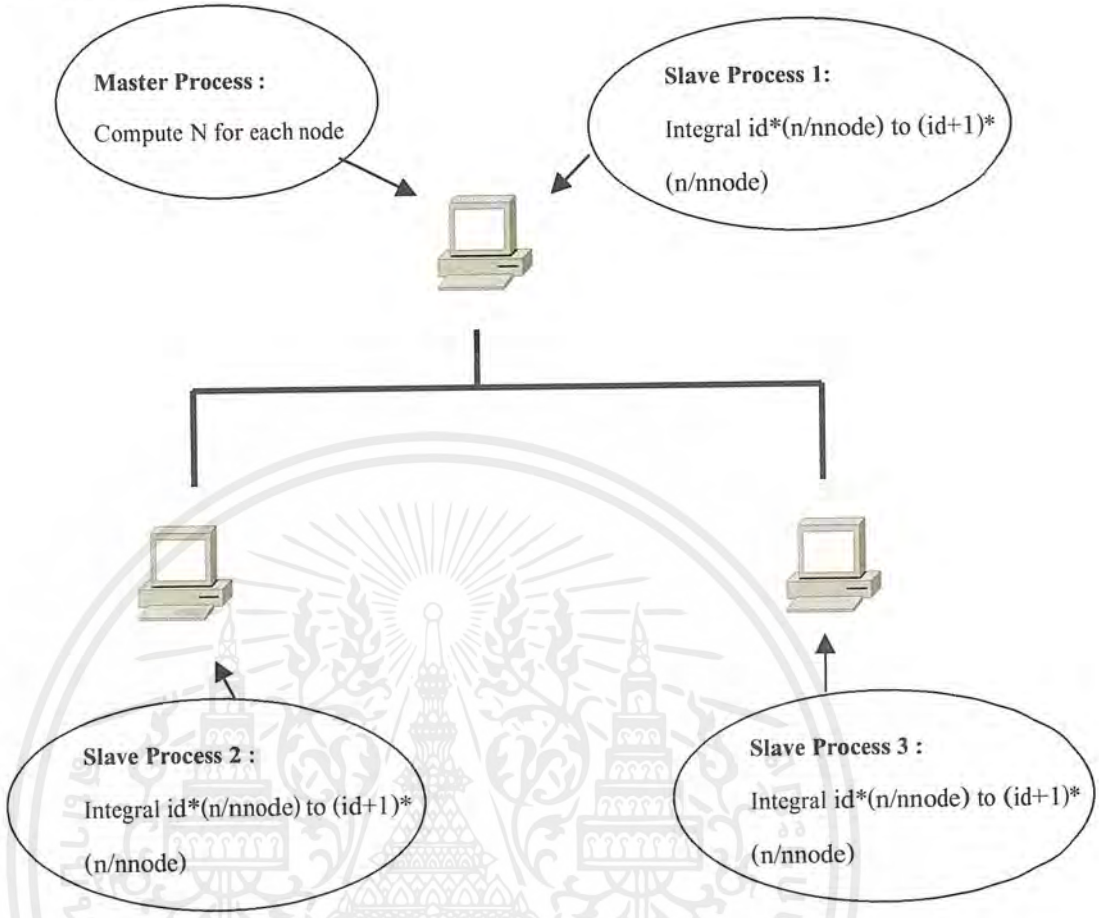
รูปที่ 9.1 แสดงลักษณะการทำงานแบบใช้เครื่องเดียวโดยใช้ไลบรารี่ PVM

9.3.1.3 โมเดลของการประมวลผลโปรแกรมที่ทำงานแบบขนานโดยใช้เครื่องพิวเตอร์ 2 เครื่องที่แต่ละเครื่องมีโพรเซสเซอร์เดียว



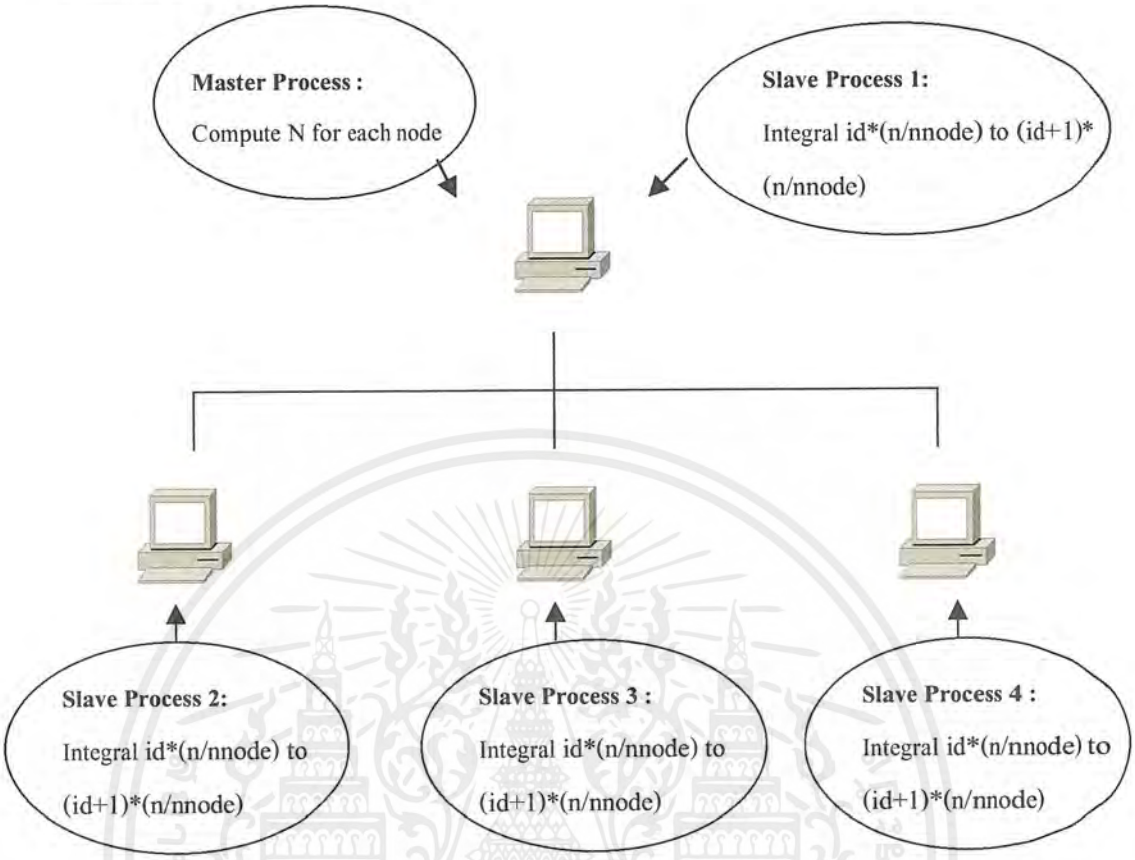
รูปที่ 9.2 แสดงลักษณะการทำงานแบบใช้เครื่อง 2 เครื่องโดยใช้ไลบรารี่ PVM

9.3.1.4 โมเดลของการประมวลผลโปรแกรมที่ทำงานแบบขนานโดยใช้เครื่องพิวเตอร์ 3 เครื่องที่แต่ละเครื่องมีโพรเซสเซอร์เดียว



รูปที่ 9.3 แสดงลักษณะการทำงานแบบใช้เครื่อง 3 เครื่อง โดยใช้ไลบรารี่ PVM

9.3.1.5 รูปแบบของการประมวลผลโปรแกรมที่ทำงานแบบขนานโดยใช้เครื่องพิวเตอร์ 4 เครื่อง ที่แต่ละเครื่องมีโพรเซสเซอร์เดียว



รูปที่ 9.4 แสดงลักษณะการทำงานแบบใช้เครื่อง 4 เครื่องโดยใช้ไลบรารี่ PVM

9.3.1.6 ผลการประมวลผลโปรแกรมบนเครือข่าย 10 M

จำนวนเครื่อง	Task ID	เวลาที่ใช้ (วินาที)	หมายเหตุ
1	T40003	20	PVM Stand Alone = 3.14159266143437276853
2	T40006 T80001	11	Parallel 3.14159266143480220279
3	T40009 T80002 TC0001	9	Parallel = 3.14159266143472271082
4	T4000C T80003 TC0002 T100001	7	Parallel = 3.14159266143467830190

9.3.1.6 ผลการประมวลผลโปรแกรมบนเครือข่าย 100M

จำนวนเครื่อง	Task ID	เวลาที่ใช้ (วินาที)	หมายเหตุ
1	T40003	19	PVM Stand Alone = 3.14159266143437276853
2	T40006 T80001	12	Parallel 3.14159266143480220279
3	T40009 T80002 TC0001	8	Parallel = 3.14159266143472271082
4	T4000C T80003 TC0002 T100001	7	Parallel = 3.14159266143467830190

9.3.1.7 วิเคราะห์ผลการศึกษาโปรแกรมคำนวณหาค่า Pi

จากผลการศึกษาทดลองประมวลผลโปรแกรมเปรียบเทียบระหว่างแบบตามลำดับ (Sequential) กับแบบขนาน (Parallel) พบว่าโปรแกรมที่ทำงานแบบขนานที่ใช้เครื่อง 1 เครื่อง ประมวลผลมีความเร็วในการทำงานที่ไม่แตกต่างกับโปรแกรมที่ทำงานแบบตามลำดับ แต่ในการประมวลผลโดยใช้โปรแกรมแบบขนานเมื่อใช้เครื่อง 2 เครื่องจะเริ่มเร็วกว่าการประมวลผลโปรแกรมที่ทำงานแบบตามลำดับ และจะเร็วมากขึ้นเมื่อใช้เครื่อง 3 เครื่องมาประมวลผลโปรแกรมแบบขนาน และจะเร็วขึ้นอีกเมื่อใช้เครื่อง 4 เครื่องมาประมวลผลโปรแกรมแบบขนาน เมื่อมาพิจารณาการทำงานของโปรแกรมแบบขนานโดยใช้เครื่อง 1, 2, 3, 4 เครื่องกันเองแล้วจะพบว่าการใช้เครื่องมากขึ้นจะทำให้โปรแกรมที่ทำงานแบบขนานทำงานเร็วขึ้นมาก แต่ถึงจำนวนเครื่อง 4 เครื่องแล้วการทำงานของโปรแกรมจะไม่เร็วขึ้นมากนักเมื่อเทียบกับเมื่อใช้ 3 เครื่อง ก็เป็นเพราะเวลาที่ใช้ในการส่งข้อมูลเมื่อมี 4 เครื่องนั้นมีผลกับการทำงานของโปรแกรมมากกว่าเวลาที่ลดลงเพราะใช้เครื่องมากขึ้น

และเมื่อพิจารณาการประมวลผลแบบขนานบนเครือข่ายที่มีความเร็ว 10 M และ 100 M พบว่าเวลาที่ใช้ในการประมวลผลมีค่าใกล้เคียงกัน

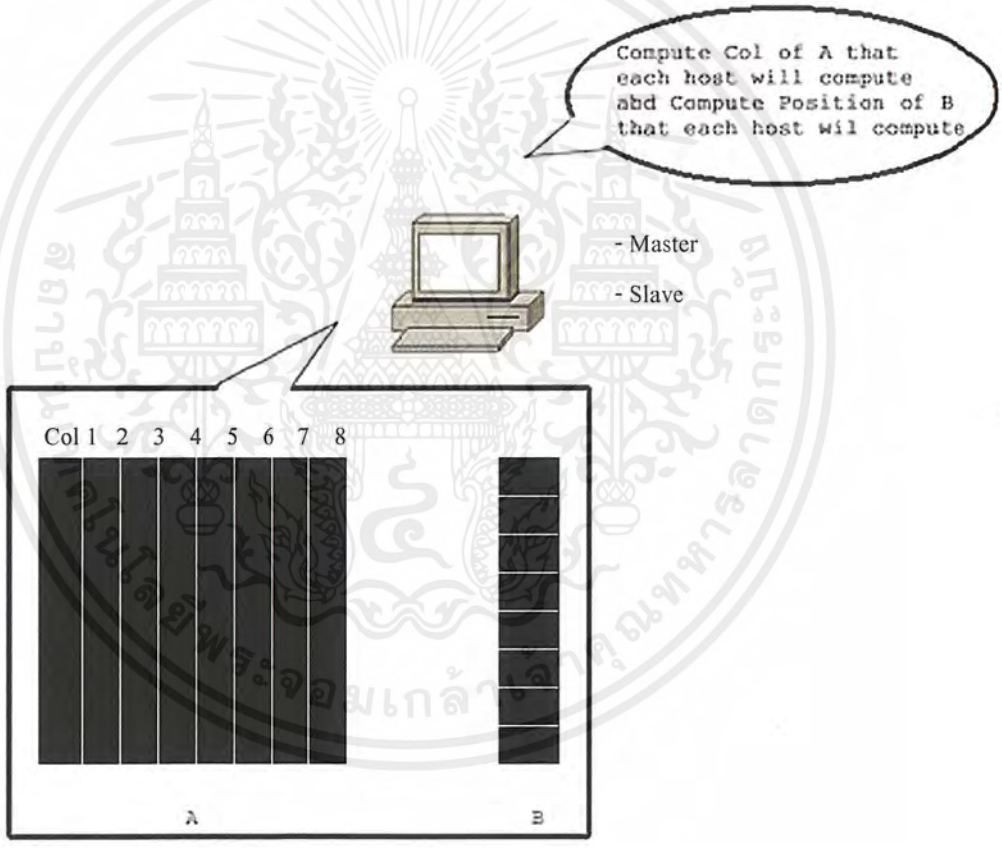
จากผลการศึกษาที่กล่าวมาแล้วทำให้พอจะสรุปได้ว่าการประมวลผลโปรแกรมแบบตามลำดับจะช้ากว่าการประมวลแบบตามขนานใช้เครื่องมากกว่า 1 เครื่อง แต่โปรแกรมทำงานแบบขนานจะประมวลผลเร็วขึ้นเรื่อยๆ เมื่อใช้เครื่องเพิ่มขึ้น แต่เมื่อใช้เครื่องถึงจำนวนหนึ่งแล้วความเร็วในการประมวลผลจะไม่เพิ่มขึ้นหรือจะเพิ่มขึ้นเพียงเล็กน้อยไม่เพิ่มขึ้นมากเหมือนการเพิ่มจำนวนเครื่องในช่วงแรกๆ เนื่องจากว่าการทำงานของโปรแกรมต้องเกิดช่วงเวลาที่เครื่องพ่อ (Master) รอผลลัพธ์จาก เครื่องลูก (Child) นั้นมากกว่าหรือเท่ากับเวลาที่ใช้ในการคำนวณที่ลดลงเพราะใช้เครื่องมากขึ้น ซึ่งเพราะเหตุนี้ทำให้การใช้เครื่อง 4 เครื่องประมวลผลได้ไม่เร็วกว่าใช้ 3 เครื่องมากนัก

9.3.2 โปรแกรมคำนวณหาผลลัพธ์ของเมทริกซ์คูณเวคเตอร์

9.3.2.1 ประมวลผลโปรแกรมคำนวณหาผลลัพธ์ของเมทริกซ์คูณเวคเตอร์ที่ทำงานแบบตามลำดับ

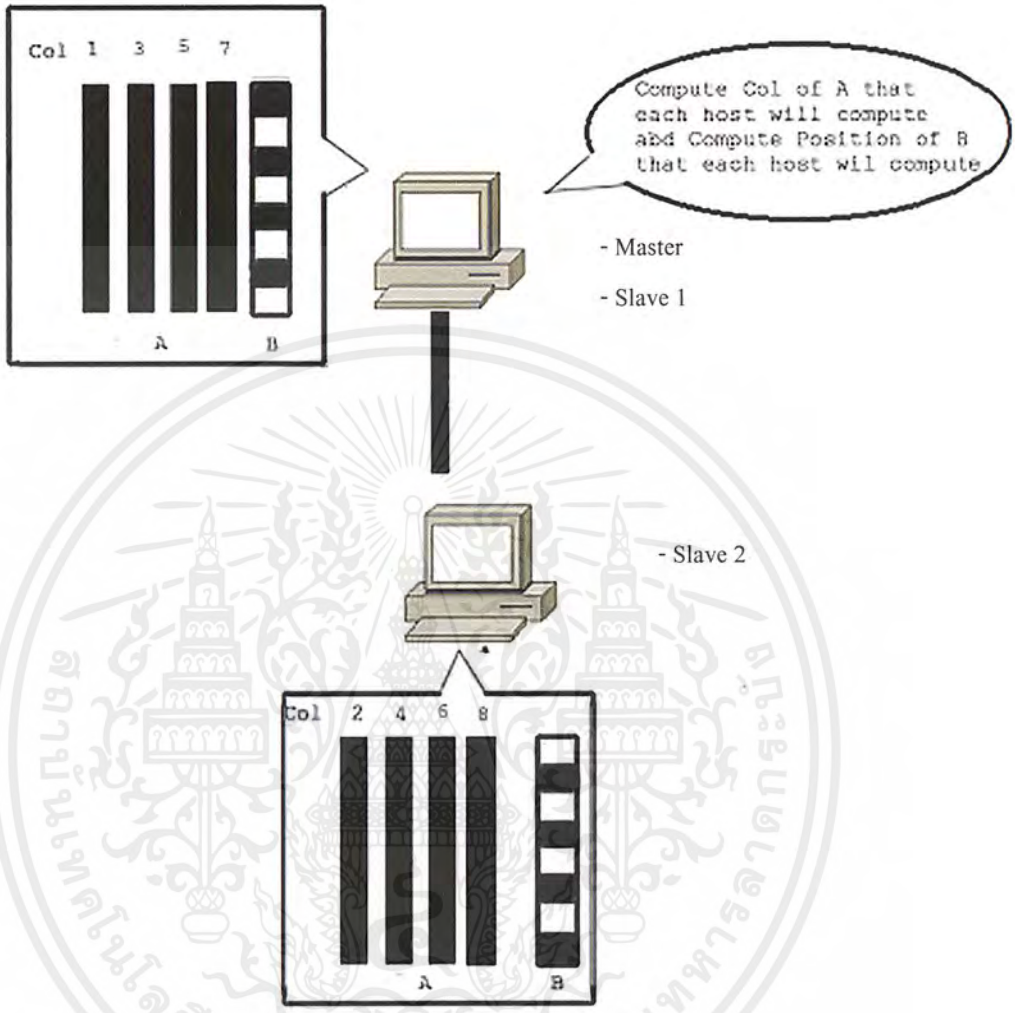
จำนวนเครื่อง	เวลาที่ใช้	หมายเหตุ
1	9	Sequential

9.3.2.2 รูปแบบของการประมวลผลโปรแกรมที่ทำงานแบบขนานโดยใช้เครื่องพิวเตอร์ 1 เครื่องที่มีโพรเซสเซอร์เดียว



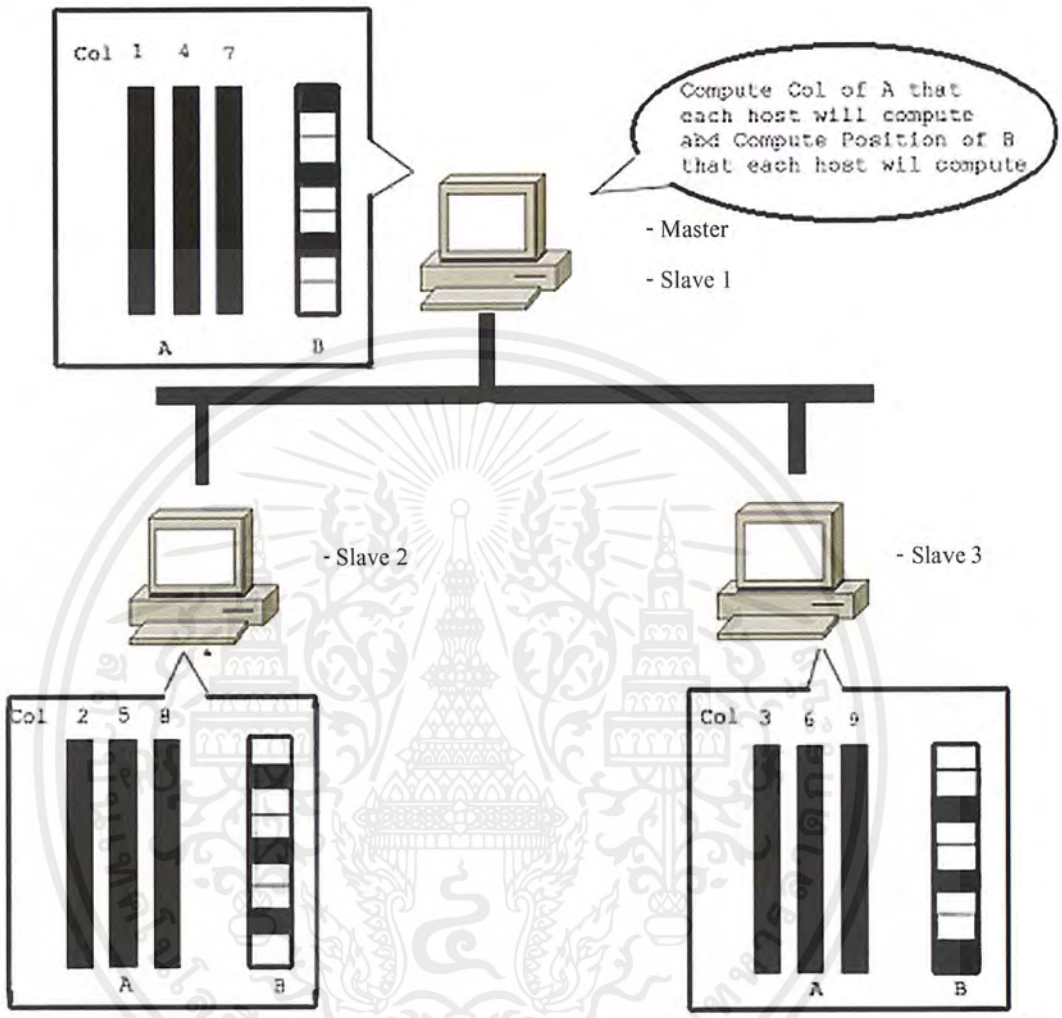
รูปที่ 9.5 แสดงลักษณะการทำงานแบบใช้เครื่องเดียวโดยใช้ไลบรารี่ PVM

9.3.2.3 โมเดลของการประมวลผลโปรแกรมที่ทำงานแบบขนานโดยใช้เครื่องพิวเตอร์ 2 เครื่องที่แต่ละเครื่องมีโพรเซสเซอร์เดียว



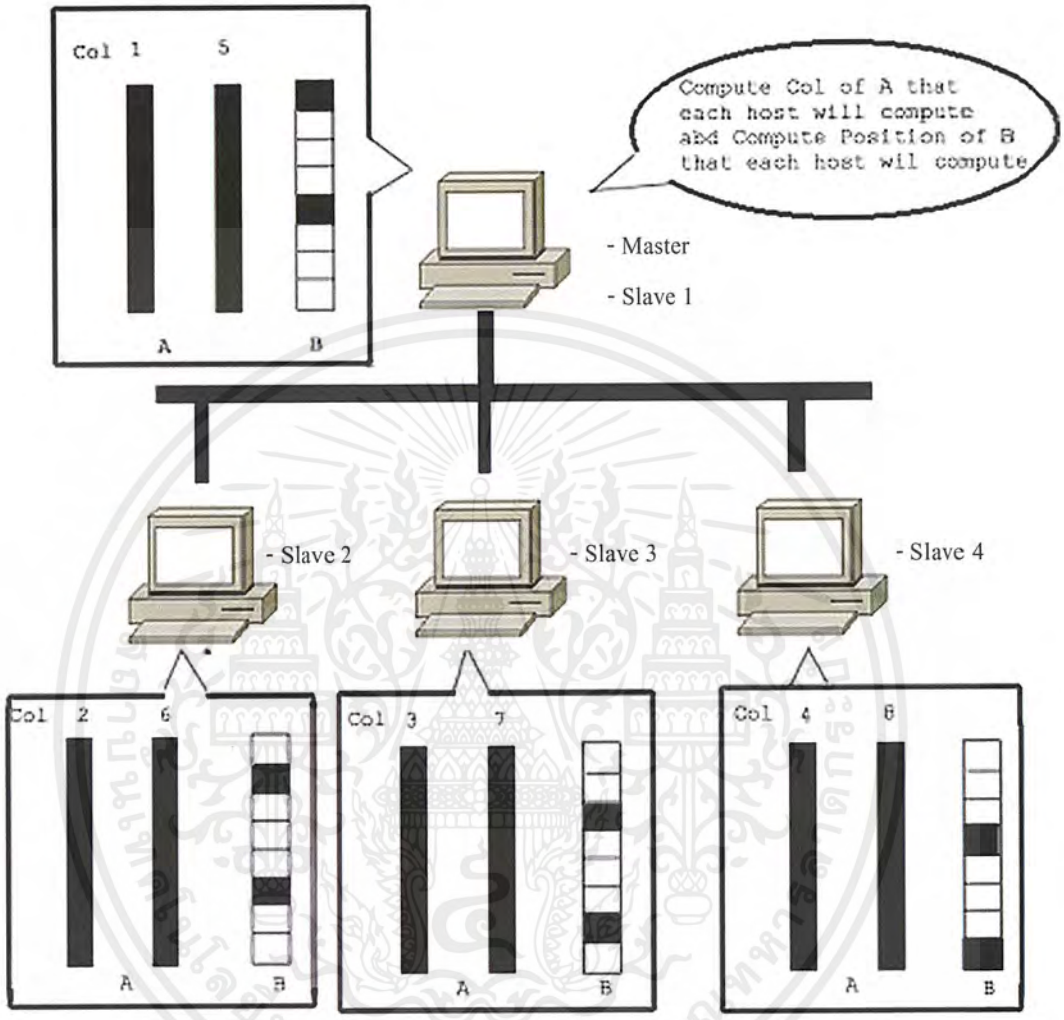
รูปที่ 9. 6 แสดงลักษณะการทำงานแบบใช้เครื่อง 2 เครื่องโดยใช้ไลบรารี่ PVM

9.3.2.4 โมเดลของการประมวลผลโปรแกรมที่ทำงานแบบขนานโดยใช้เครื่องพิวเตอร์ 3 เครื่องที่แต่ละเครื่องมีโพรเซสเซอร์เดียว



รูปที่ 9.7 แสดงลักษณะการทำงานแบบใช้เครื่อง 3 เครื่อง โดยใช้ไลบรารี่ PVM

9.3.2.5 โมเดลของการประมวลผลโปรแกรมที่ทำงานแบบขนานโดยใช้เครื่องพิวเตอร์ 4 เครื่องที่แต่ละเครื่องมีโพรเซสเซอร์เดียว



รูปที่ 9.8 แสดงลักษณะการทำงานแบบใช้เครื่อง 4 เครื่องโดยใช้ไลบรารี่ PVM

9.3.2.6 ผลการประมวลผลโปรแกรมบนเครือข่าย 10 M

จำนวนเครื่อง	Task ID	เวลาที่ใช้ (วินาที)	หมายเหตุ
1	T40003	9	PVM Stand Alone N=3000
2	T40006 T80001	238	Parallel N=3000
3	T40009 T80002 TC0001	353	Parallel N=2997 *
4	T4000C T80003 TC0002 T100001	429	Parallel N=3000

9.3.2.6 ผลการประมวลผลโปรแกรมบนเครือข่าย 100M

จำนวนเครื่อง	Task ID	เวลาที่ใช้ (วินาที)	หมายเหตุ
1	T40003	7	PVM Stand Alone N=3000
2	T40006 T80001	123	Parallel N=3000
3	T40009 T80002 TC0001	156	Parallel N=2997 *
4	T4000C T80003 TC0002 T100001	258	Parallel N=3000

9.3.2.7 วิเคราะห์ผลการศึกษาโปรแกรมคำนวณหาผลลัพธ์ของเมทริกซ์คูณเวคเตอร์

จากผลการศึกษาทดลองประมวลผลโปรแกรมเปรียบเทียบระหว่างแบบตามลำดับ (Sequential) กับแบบขนาน (Parallel) พบว่าโปรแกรมที่ประมวลผลแบบขนานที่ใช้เครื่อง 1 เครื่องประมวลผลมีความเร็วในการประมวลผลที่ไม่แตกต่างกับโปรแกรมที่ประมวลผลแบบตามลำดับ แต่ในการประมวลผลโดยใช้โปรแกรมแบบขนานเมื่อใช้เครื่อง 2 เครื่องจะช้ากว่าโปรแกรมที่ประมวลผลแบบตามลำดับกับโปรแกรมแบบขนานที่มีใช้เครื่องๆเดียว และจะยิ่งช้าลงมากขึ้นเมื่อใช้เครื่อง 3 เครื่องมาประมวลผลโปรแกรมแบบขนาน และจะช้าลงอีกขึ้นอีกเมื่อใช้เครื่อง 4 เครื่องมาประมวลผลโปรแกรมแบบขนาน

เมื่อมาพิจารณาการประมวลผลของโปรแกรมแบบขนานโดยใช้เครื่อง 1, 2, 3, 4 เครื่อง แล้วจะพบว่าการใช้เครื่องมากขึ้นจะทำให้โปรแกรมที่ทำงานแบบขนานทำงานช้าลง ยิ่งเพิ่มเครื่องขึ้นก็จะยิ่งใช้เวลาในการประมวลผลช้าลงมากขึ้นเรื่อยๆ

และเมื่อพิจารณาการประมวลผลแบบขนานบนเครือข่ายที่มีความเร็ว 10 M และ 100 M พบว่าเวลาที่ใช้ในการประมวลผลของโปรแกรมทำงานบนเครือข่าย 100 Mจะใช้เวลาในการประมวลผลเร็วกว่าโปรแกรมทำงานบนเครือข่ายที่มีความเร็ว 10 M

จากผลการศึกษาที่กล่าวมาแล้วทำให้พอจะสรุปได้ว่าการประมวลผลโปรแกรมแบบตามลำดับจะใช้เวลาในการประมวลผลเท่าๆกับเวลาที่ใช้ในโปรแกรมแบบขนานที่ทำงานโดยใช้เครื่อง 1 เครื่อง และจากผลการประมวลผลที่ใช้จำนวนเครื่องมากกว่าแล้วทำให้เวลาที่ใช้ในการประมวลผลกลับเพิ่มมากขึ้น อาจจะเป็นไปได้ว่าโปรแกรมที่มีการส่งข้อมูลที่มีขนาดใหญ่จะทำให้เกิดเวลาที่เสียเปล่า (Overload time) ในการส่งข้อมูลระหว่างกัน ทำให้เป็นผลกระทบกับเวลาส่วนใหญ่ที่ลดลงเนื่องจากใช้หลายเครื่องช่วยกันประมวลผล

บทที่ 10
สรุปผลการศึกษาและข้อเสนอแนะ

10.1 สรุปผลการศึกษา

จากการศึกษาและได้ทำปัญหาพิเศษเรื่อง "กรรมวิธีการประมวลผลแบบขนานบนพีซีคลัสเตอร์" (Parallel Processing Method on PC Cluster) ซึ่งได้ใช้ระบบปฏิบัติการลีนุกซ์ (LINUX) เป็นระบบปฏิบัติการประจำเครื่องที่ใช้ทดสอบ ทำให้ได้รู้การทำงานต่างๆ ของระบบปฏิบัติการนี้ไม่ว่าจะเป็นด้านการกำหนดการแสดงผล, การติดตั้งการ์ดเครือข่าย (Network Card), การกำหนดและการปรับแต่งทางด้านระบบเครือข่ายของระบบปฏิบัติการนี้

สำหรับไลบรารี่ PVM นี้การติดตั้งไลบรารี่นี้ทำให้ได้ทราบการคอมไพล์และการทำงานของโปรแกรมที่ทำงานบนระบบปฏิบัติการนี้, ทำให้ได้ทราบการทำงานเกี่ยวกับการใช้รีโมตเชลล์ (Remote Shell) บนระบบปฏิบัติการลีนุกซ์

ด้านการทำงานของพีซีคลัสเซอร์ทำให้ทราบหลักการและลักษณะการทำงานต่างๆ ของพีซีคลัสเตอร์

ในด้านการศึกษาหลักการประมวลผลของโปรแกรมแบบขนาน ทำให้ทราบการจัดแบ่งงานและรูปแบบของโปรแกรมแบบขนาน และยังทำให้ทราบลักษณะการเขียนโปรแกรมแบบขนาน

สำหรับการทำงานในการเขียนและการทดสอบโปรแกรมที่ประมวลผลแบบขนานก็ทำให้พบตัวแปรต่างๆ และผลกระทบที่ทำให้การประมวลผลของโปรแกรมแบบขนานมีความเร็วขึ้น นั่นก็คือจำนวนข้อมูลที่มีการแลกเปลี่ยนกันถ้ามากการส่งข้อมูลจำนวนมากหรือมีขนาดใหญ่ระหว่างกัน จะทำให้เสียเวลาในการทำงานในส่วนนี้มาก และความเร็วของระบบเครือข่ายก็มีผลการเวลาที่ใช้ในการประมวลผลเมื่อมีการส่งข้อมูลกันมากๆ ถ้าใช้ระบบเครือข่ายที่มีความเร็วต่ำก็จะเกิดช่วงเสียเวลา (Overload time) ซึ่งทำให้เวลาที่ใช้ในการประมวลผลโปแกรมแบบขนานบนเครือข่ายความเร็ว 10 M ใช้เวลามากว่า 100 M

10.2 ปัญหาในการทำปัญหาพิเศษ

- ปัญหาในการติดตั้งระบบปฏิบัติการลีนุกซ์ เนื่องจากระบบปฏิบัติการนี้ใช้ระบบไฟล์เป็นแบบ Ext2 จึงจำเป็นต้องมีการฟอร์แมตพาติชั่นให้เป็นแบบ Ext2
- ปัญหาที่เกิดจากการปรับแต่งและการติดตั้งอุปกรณ์ต่างๆ เนื่องจากไดร์เวอร์ของอุปกรณ์บางชนิดยังไม่สนับสนุนระบบปฏิบัติการนี้ทำให้ต้องมีการดาวน์โหลดจากเวบไซต์ผู้ผลิตเพื่อมาติดตั้ง ซึ่งการติดตั้งก็มีความยุ่งยากมาก
- ปัญหาจากการเขียนโปรแกรม เนื่องการเขียนโปรแกรมแบบขนานต้องมีการแตกงานหรือแตกช่วงการคำนวณเป็นส่วนๆ เพื่อให้เครื่องต่างๆ ช่วยกันประมวลผล ในส่วนนี้เองทำให้ยากในการคิดและเขียนโปรแกรม

10.3 ข้อเสนอแนะ

- ควรที่จะมีการทดสอบการประมวลผลโปรแกรมที่มีการส่งข้อมูล (Message) กันมากๆ เพราะที่จะหาผลกระทบถึงจำนวนข้อมูล (Message) ที่มีการส่งว่ามีผลกระทบกับเวลาที่ใช้ในการประมวลผลหรือไม่
- ควรที่จะมีการทดสอบการประมวลผลโปรแกรมโดยใช้จำนวนเครื่องที่มากขึ้นเพื่อหาผลกระทบของจำนวนเครื่องต่อเวลาที่ใช้ในการประมวลผล เพราะปัญหาพิเศษนี้ใช้จำนวนเครื่องเพียง 4 เครื่องเท่านั้น
- ควรทดสอบการเขียนโปรแกรมโดยใช้ไลบรารี่อื่นๆ เช่น MPI เพื่อเปรียบเทียบประสิทธิภาพการทำงานกับโปรแกรมที่ใช้ไลบรารี่ PVM
- ควรทดสอบเขียนโปรแกรมที่ทำงานบนเครื่องมัลติโพรเซสเซอร์และใช้คอมไพเลอร์ที่สนับสนุนการประมวลผลแบบขนาน เพื่อเปรียบเทียบประสิทธิภาพการทำงานกับโปรแกรมที่ทำงานโดยใช้ไลบรารี่ PVM

ภาคผนวก ก

Setup PVM

เหตุผลหนึ่งที่ PVM เป็นที่นิยมนั้นเพราะว่า setup และใช้งานง่าย PVM ไม่ต้องใช้การติดตั้งพิเศษ แค่ใช้ Login บน Host ก็สามารถทำได้ เพียงแค่ทุกเครื่องที่จะใช้ติดตั้ง PVM ซึ่ง PVM ใช้ ตัวแปร 2 ตัว เพื่อเริ่ม start และ run โดยแต่ละ PVM user จำเป็นต้อง set 2 ตัวแปรนี้ในการใช้ PVM ตัวแปรแรก คือ PVM ROOT , ซึ่ง set ที่ไดเรคทอรี่ pvm3 ตัวแปรที่สอง คือ PVM_ARCH, ซึ่งบอกถึง architecture ของ host ถึงจะสามารถ execute จาก PVM_ROOT ไดเรคทอรี่

วิธีที่ง่ายที่สุด คือ set 2 ตัวแปรนี้ใน file ".cshrc" ซึ่งเราให้ใช้ csh (คอร์นเชลล์) นี่คือตัวอย่างการ set PVM_ROOT : setenv PVM_ROOT $HOME/pvm3

ส่วนการ set PVM_ARCH โดยการนำรายละเอียดของ file $PVM_ROOT/lib/cshrc.stub มาต่อท้าย file .cshrc โดย stub จะอยู่หลัง PATH และ PVM_ROOT ที่ถูกกำหนดไว้แล้ว stub นี้จะเลือก PVM_ARCH สำหรับ host โดยอัตโนมัติและจะมีประโยชน์มากสำหรับผู้ใช้ ที่ share file system (เช่น NFS) ตามแต่ละ architecture

การสร้างแต่ละประเภท architecture จะทำโดยอัตโนมัติโดย logging เข้าไปบน host และไปที่ PVM_ROOT ไดเรคทอรี่แล้วพิมพ์ make ซึ่งจะหา architecture ที่กำลัง execute อยู่โดยอัตโนมัติ และสร้างไดเรคทอรี่ย่อย และสร้าง file pvm, pvmd3, libpvm3.a, libfpvm3.a, pvmgs, libgpvm3.a ซึ่งจะอยู่ใน ไดเรคทอรี่ $PVM_ROOT/lib/PVM_ARCH ยกเว้น pvmgs จะอยู่ใน $PVM_ROOT/bin/PVM_ARCH

สรุปการ setup

- set PVM_ROOT และ PVM_ARCH ในfile .cshrc
- สร้าง PVM สำหรับแต่ละประเภท architecture
- สร้าง file .rhosts บนแต่ละ host ซึ่งเป็นรายชื่อ host ทั้งหมดที่ใช้

Starting PVM

ก่อนที่จะ compile และ run parallel PVM programs จะต้องแน่ใจว่าสามารถ start PVM และ configure virtual machine โดยพิมพ์

```
%pvm
```

และจะได้รับ console prompt ซึ่งจะบ่งบอกว่า PVM running บน host นี้คุณสมารถ add host บน virtual machine โดยพิมพ์ที่ PVM prompt ดังนี้

```
pvm> add hostname
```

และสามารถลบ host (ยกเว้นเครื่องที่คุณ run pvm อยู่) จาก virtual machine โดยพิมพ์

```
pvm> delete hostname
```

ถ้าได้รับ message ว่า " Can't Start pvmd " เมื่อตรวจปัญหาที่เกิดขึ้นและลองดูอีกครั้งจะเห็นสิ่งที่แสดงถึง virtual machine โดยพิมพ์

pvm> conf

และเห็นว่างานใดที่กำลัง run อยู่บน virtual machine โดยพิมพ์

pvm> ps –a

ถ้าพิมพ์ "quit" ที่prompt จะออกจาก PVM console แต่ virtual machine และงานต่างๆ ยังคง run ต่อไปที่ prompt ปกติบน host สามารถพิมพ์ %pvm และจะมี message " PVM already running " และเข้า PVM prompt เมื่อต้องการจะจบการทำงานของ virtual machine โดยพิมพ์

pvm> halt

โดยคำสั่งนี้จะ kill บาง PVM tasks และ shutdown virtual machine และออกจาก PVM console ซึ่งควรจะ stop PVM ก่อนเพราะ จะทำให้แน่ใจได้ว่า virtual จะ shutdown ได้หมด

ถ้าไม่ต้องการจะพิมพ์ hostname ทีละครั้ง โดยให้ใช้ hostfile ซึ่งจะเก็บ list hostname ไว้หนึ่งบรรทัดต่อหนึ่ง host

%pvm hostfile

PVM จะเพิ่ม host ทั้งหมดก่อนconsole prompt ปรากฏ

บรรณานุกรม

Piyush J. Hatcher and Michael J. Quinn 1991. **Data-Parallel Programming on MIMD Computer**

Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam 1994 **PVM : Parallel Virtual Machine, A user guide and tutorial for Network Parallel Computing**

Kai Hwang, Zhiwei Xu **Scalable parallel computing : technology, architecture, programming / Kai Hwang, Zhiwei Xu**

Prithviraj Banerjee ,**Parallel algorithms for VLSI computer-aided design**

E.V. Krishnamurthy , **Parallel processing : principles and practice E.V. Krishnamurthy**

Netlib Repository at UTK and ORNL, "**Index for PVM3 Library**" [Online], Available : http://www.netlib.org/pvm3/index.htm

Parallel Computing resource " **PVM, Parallel Virtual Machine**" [Online] , Available : http://www.epm.ornl.gov/