# 個人電腦叢集中光跡追蹤成像法之平行化
# Parallelization of Ray Tracing in PC Clusters

盧能彬
長庚大學資訊管理學系
E-mail: nplu@mail.cgu.edu.tw

許勝涵
長庚大學資訊管理學系
E-mail: m8944010@stmail.cgu.edu.tw

## 摘要

本論文藉由電腦叢集，探討光跡追蹤成像法的平行化處理。我們建置個人電腦叢集，進行光跡追蹤成像法的加速測量。整體系統環境如下：個人電腦叢集為平行處理平台，作業系統部分為 Redhat Linux 6.2，編譯器是 gcc 2.91.66。平行程式發展工具是 PVM 3.4.3，平行模式為訊息傳送 (Message Passing)，平行演算法採行程場 (Process Farm)，所選擇的光跡追蹤軟體是 POVRAY3.1。經使用行程場平行演算法後，加速度已有近似線性比率的提昇。但因仍存在一些負載不平衡的問題，故最後我們提出一可改善負載平衡的方法，經實驗證明，我們的方法可再增加平均約 8.3%的效能。

**關鍵詞**：個人電腦叢集，平行程式設計，光跡追蹤法。

## Abstract

With PC clusters, this research parallelized the mature high-speed ray-tracing algorithms to further boost the rendering speed. The details of our parallel processing system are as follows. The hardware is the PC cluster and network devices. On top of the hardware is system software, including operating system and compiler. The operating system is Redhat Linux 6.2, and the compiler is gcc 2.91.66. We use PVM 3.4.3 as our parallel programming tool and the parallel model is message passing. Our parallel algorithm adopts the process farm model and the application parallelized is the popular ray tracing software: POVRAY 3.1. From the parallelization study in PC clusters, we got near linear speedup. But there still were some load imbalance problems in the system. Therefore, we proposed an approach to improve the system load balance. The experimental results proved it could increase about 8.3 percents efficiency further in average.

**Keywords:** PC Cluster, Parallel Programming, Ray Tracing.

## 一、緒論

1994 年美國太空總署 (NASA) 的 CESDIS (Center of Excellence in Space Data and Information Science ) 為了進行地球與太空科學計畫，嘗試用低廉而易得的電腦相關零件 (Commodity Off The Shelf)，來組裝大量平行電腦，以應付該計畫所需處理的大量計算。計畫初期，Thomas Sterling 和 Don Becker 將十六個 Intel 100 MHz DX4，以 10 Mbits/s Ethernet 組裝成一個叢集，並取名為 Beowulf [2] 。之後 Beowulf Project 便很快地蔓延到 NASA 的其他單位以及美國的其他研究單位。時至今日，世界各地許多個人電腦叢集，都可歸屬於 Beowulf Cluster [3, 4, 6, 8, 9, 13]。

個人電腦叢集目前已廣泛被應用在許多商業應用、科學計算領域上，商業應用包括如資料探勘、網路伺服器、地理資訊系統、資料庫等；科學計算包括如核子試爆模擬、基因工程、地球結構模擬、大氣動力模擬、計算機圖學等等。在計算機圖學領域中有不少需大量計算的成像法，光跡追蹤法 (Ray Tracing) 即屬於此種成像法，其雖可以描繪出相當真實的 3D 影像，但也由於需做各光線與全部物體的交差判定及找出最靠近視點的交點，故常會耗費大量 CPU 計算時間。過去已有不少方法被提出用來改善光跡追蹤法之成像速度，其重點大都在減少交差判定的總次數及交差判定處理本身高速化上。本研究則建立在已發展成熟的加速演算法上，更進一步的發展出對光跡追蹤的平行處理方法，並藉由加速 (Speedup) 的提昇來證實個人電腦叢集的優越性能。

本研究之系統架構如圖 1 所示，可分為六個部分：在硬體實驗平台部分，我們使用了二至六台電腦並以 100Mbps 的 DLink DES-1008D Switch 連結之，在系統軟體方面，作業系統為 Redhat Linux 6.2 based on kernel 2.2.14 [12]，程式編譯器及除錯器採 gcc 2.91.66 及 gdb 4.18。使用的平行程式函式庫為屬於訊息傳送 (Message Passing) 的 PVM (Parallel Virtual Machine) 3.4.3 [11] 並輔以 XPVM

1.2.5 [14] 作為行程執行時間訊息傳送過程之圖形化觀察工具。本研究的平行演算法採行程場 (Process Farm) [7]，光跡追蹤軟體則是目前在 Unix 和 Windows 平台頗著名的 POVRAY3.1 (Persistence Of Vision RAY tracer) [10]。

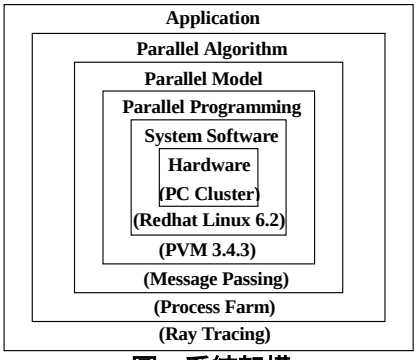


**圖 1 系統架構**

本論文共分為六個章節。在第一章的緒論中，介紹本研究的動機與目的。第二章為 POVRAY 循序程式流程分析，藉由分析 POVRAY 的控制流程和資料流程來解析程式，進而尋找可行的平行化方式。第三章為 POVRAY 平行化方法，主要是說明如何將 POVRAY 平行化，並評估平行化後的加速 (Speedup)。第四章為負載平衡改善，討論如何將第三章平行方法的一些缺點做改善以增加系統的效能，第五章則為本論文的總結。

## 二、POVRAY 循序程式流程分析

要平行化一個軟體，首要是先了解其程式的整個流程，在一步一步追蹤程式的過程中，找出之所以需大量執行時間的程式碼所在，再設法與予平行化。通常考慮到的程式流程不外乎控制流程 (Control Flow) 及資料流程 (Data Flow) 兩種，因此我們將依序分析 POVRAY 的控制流程和資料流程。

### 2.1 控制流程

一開始程式會做一些記憶體配置，如 text stream 的記憶體配置的動作，接著會解析使用者所下的命令參數以便做適當的回應，接下來初始化與材質係數相關的 Noise Table，若執行 shell-out command (ini 檔中的使用者命令) 沒有任何錯誤，則開始 FrameRender()，否則結束 POVRAY 程式。描繪完成後會列印一些狀態及統計資訊，包括程式執行時間、記憶體用量、像素總數等 (圖 2)。

FrameRender() 副程式為整個計算過程的核心 (圖 3)，此副程式會先去解析場景檔及準備建構場景模型之資料結構，並 Open Output File 和初始化動作，Open Output File 是為因應接續前次之 Render 結果，如果 Radiosity Pre-view 已經完成則直接做 Non_Adaptive_Tracing()，否則視 Radiosity 選項是否開啟做 Tracing_Mosaic_Smooth 或 Preview，Tracing 之後關閉檔案並列印資訊。

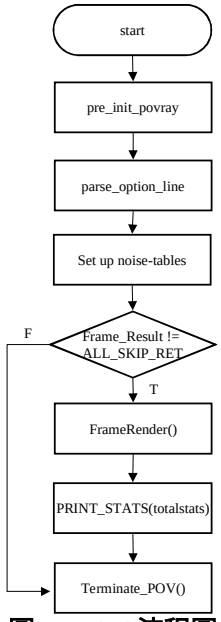


**圖 2 main()流程圖**

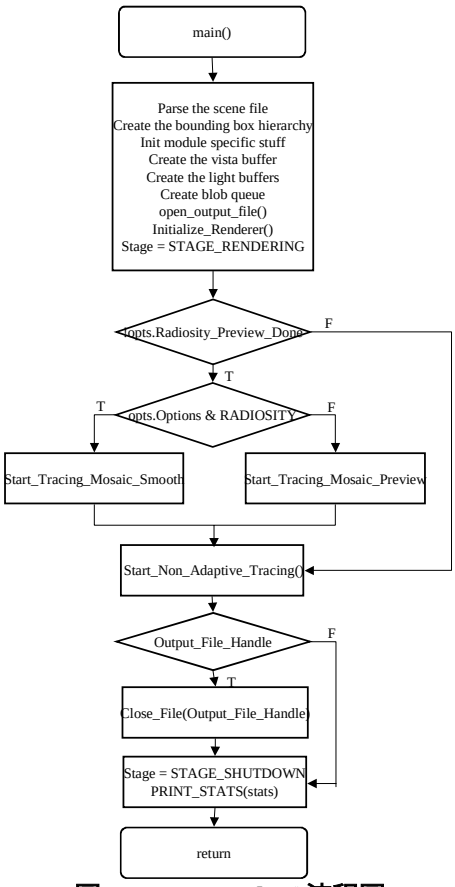


**圖 3 FrameRender()流程圖**

Non_Adaptive_Tracing()是由第一條 Line 的第一個 Column 開始，也就是由上至下由左至右 Trace 每一個像素 (Pixel)，Trace 過程中

會 Check 目前正在 Trace 第幾條 Line 和累計所用時間，並調整由許多模型物件組成的 Vista Tree，若物件的某部分不在目前所 Trace 的 Line，則會將其遮掩 (Prune)。每完成一條 Line 會將其寫入磁碟再進行下一條 Line 直到最後一條 Line (圖 4)。其中副程式 trace_pixel()會針對單一像素 (Pixel) 做處理，Trace 有用到 Focal Blur 及 Vista Buffer 的主要光線，Trace 完成後像素的顏色值會被剪下，並增加用來計算已完成像素數的計數 Counter (圖 5)。而另一個副程式 Prune_Vista_Tree()所輸入的是一條 Line。它會將所有不在目前所 Trace 的 Line 的 Nodes 標記起來，例如 Vista Tree 的 Root Node 不在目前的 Line，就將其修剪掉，反之則保留，同理也需檢查 Leaf Nodes 和其他 Siblings 是否需修剪或保留 (圖 6)。

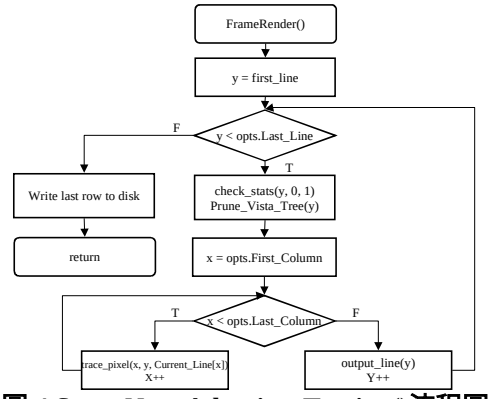


**圖 4 Start_Non_Adaptive_Tracing()流程圖**

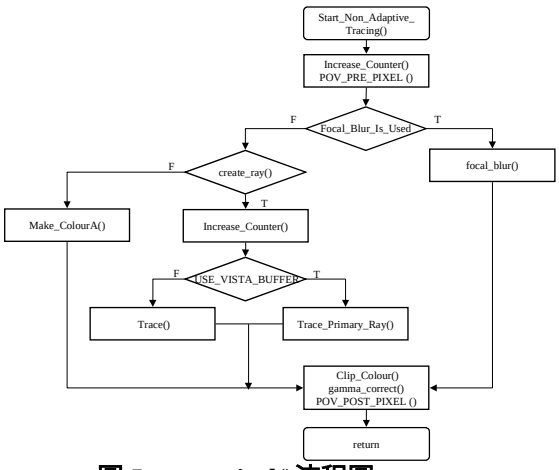


**圖 5 trace_pixel()流程圖**

## 2.2 資料流程

圖 7 為 POVRAY 之資料流程(Data Flow)。Input 包括了使用者所下的命令(template_command)、環境參數設定檔 (ini file)、場景模型檔 (pov file)，經過 POVRAY 程式處理後，輸出每個像素的 red、green、blue、alpha grey 四個顏色值，並合成最後的圖檔，檔案格式可選擇 tga、png 或 ppm，最後再輸出統計資訊，包括程式執行時間、記憶體使用量、像素總數、發出光線總數、光線與各形狀交點測試成功數及成功比例等。

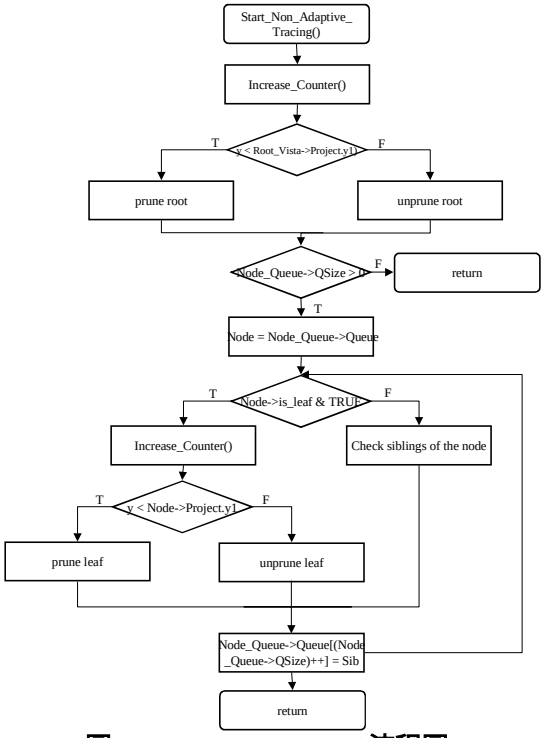


**圖 6 Prune_Vista_Tree ()流程圖**

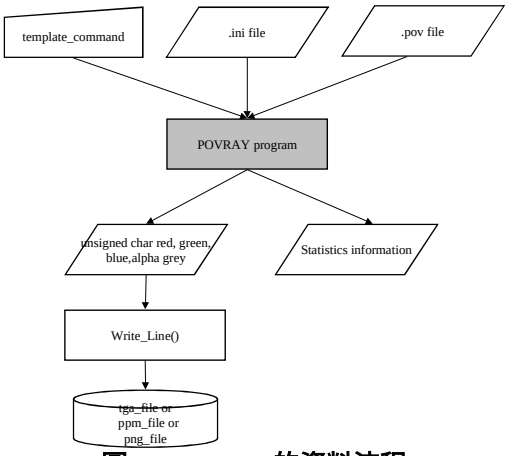


**圖 7 POVRAY 的資料流程**

在對 POVRAY 有了基本的了解後，接著便是逐步執行程式，找出之所以需大量執行時間的問題所在。經過實際的逐步追蹤與實驗，我們發現剛開始的 Parse Option 或 Scene File 及一些初始化資料結構的動作並不會花費太多時間，且時間大都是定數。真正要耗費大量時間的動作應是在描繪 (Render) 部分，在 Start_Non_Adaptive_Tracing() (圖 4) 中第一個迴圈是描繪一張圖像的 y 軸也就是寬的部分，第二個迴圈是圖像的 x 軸也就是高的部分。要描繪出一張圖一定要描繪寬度乘以高度的像素 (Pixel) 數量， 以一張 640*480 解析度的圖而言就要描繪 307200 個像素，加上每一次的迴圈都可能要做交差判定等需要複雜

數學計算的動作，故幾十萬次的迴圈有可能一跑就是幾十分鐘甚至數個小時，時間的長短還須視圖像中物件的多寡和複雜度而定。在此將 POVRAY 全部流程按其順序分為三個部份，並分別量測執行間。第一個部分為 main()中的 pre_ini_povray 到 setup noise-tables；第二個部分為 FrameRender()中的 Parse the scene file 到 Start_Tracing_Mosaic_Smooth() 或 Start_Tracing_Mosaic_Preview()；第三部份是 Start_Non_Adaptive_Tracing()：

$$T_{total} = T_{part1} + T_{part2} + T_{part3} \quad (1)$$

在此以 POVRAY 中的 chess2.pov (解析度 320*240) 為例，共花 196.873 秒完成作業 ($T_{total}$ =196.873，$T_{part1}$ = 0.005，$T_{part2}$ = 0.131，$T_{part3}$ = 196.737)。由上例可知，第一和第二部份所做的準備資料結構、建構場景等動作所耗時間均極少且大都差不多，其餘大約 99%的時間都花在第三部份 Start_Non_Adpative_Tracing()中的兩個描繪迴圈，根據 Amdahl's Law [1]，如何減少那 99%的時間將是重點，也是下一節 POVRAY 平行化的關鍵。

## 三、POVRAY 平行化

### 3.1 平行程式設計

依 Foster 所提出的平行方法，我們可以將本研究的平行程式設計分為四個步驟：切割、通訊考量、凝聚、對應[5]。在切割部分，所探討的是本論文使用的工作量分派策略、平行演算法等議題；在通訊方面是通訊時間合理化與顆粒度等問題；凝聚是結果資料回收之同步化問題與細部參數調整；對應則是程式完成後的實際執行與效能評估。茲分四小節詳述如下。

#### 3.1.1 切割 (Partitioning)

本論文是採用的是資料分解 (Data Decomposition)來進行程式的平行化。要被分解的資料就是指由像素 (Pixel) 所構成的圖像，也就是將圖像切成許多長寬相等的區塊 (Block) 分由在不同處理器上來執行以加快速度。由上一節的 POVRAY 分析，可得知每一個像素 (Pixel) 都是獨立的，也就是要描繪一個像素之前，並不需要前一個或前幾個像素的執行結果。此外在光跡追蹤過程中，除非是處理極端複雜的場景模型，否則描繪一個像素，大多只是幾百個 Microseconds 或 Milliseconds，這對於 CPU 而言或許是頗長的計算時間，但站在平行處理的角度，要將不到一秒鐘的計算量平行化，不但困難度高，且因牽涉到一些要用網路傳送訊息等因素的額外負擔，極可能達到反效果，也就是平行化後的速度會比平行化之前還慢。基於以上兩點，我們可以將顆粒度 (Granularity) 放大，把資料切割之目標放在由若干行 (Column) 和若干列 (Row) 所組成的區塊 (Block) 上，而非單一像素的切割。

有了以上基本概念後，接著將更加詳細說明我們如何切割工作及如何將行程場 (Process Farm) 對應到 POVRAY。當程式開始執行時，先會產生一主控行程 (Master Control Process)，主控程式會依叢集的節點數量來產生行程 (Fork Process)，每一個從屬節點 (Slave Node) 包括主控程式本身所在的節點都會產生一個行程，用以負責描繪像素 (Render Pixel) 和通訊，在所有的行程被產生完畢後，各行程開始讀取並解析 ini 及 pov file 以建構場景模型，因此每個節點都必需擁有一份 ini 及 pov file，或者使用網路檔案系統(NFS)分享 ini 及 pov file。完成後各行程便開始向主控程式要求工作，此時主控程式會將尚未分派出的區塊範圍 (Block Scope) 分派給要求工作的行程，之後把被分派到的區塊範圍分解成數條單一線 (Line) 丟入 POVRAY 程式執行，當做完一整個區塊後，再將整塊區塊結果顏色值包裝後傳回，由主控程式逐一條線 (Line)、逐一個像素 (Pixel) 解包裝取得顏色值，寫入磁碟組成影像檔 (圖 8)。

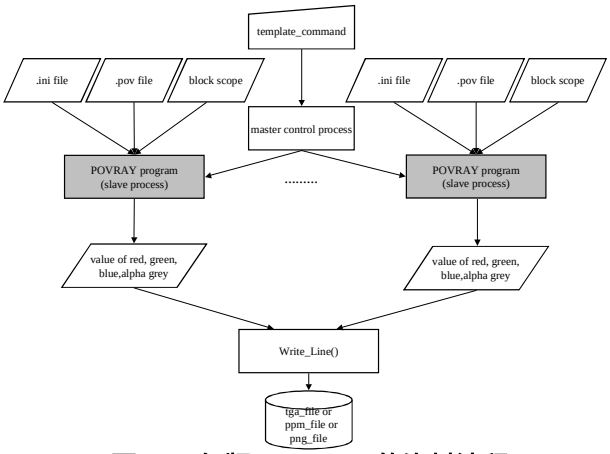


**圖 8 平行版 POVRAY 的資料流程**

另外在工作分派機制上，我們採取的是需求導向 (Demand Driven) 做法，意即工作並非由主控行程自己主動分派給從屬行程，而是當從屬行程向主控行程發出請求後，主控行程才會將工作分派出去。基本上主控行程是處於被動狀態，當從屬行程有需求時才會開始分派，待從屬行程做完工作後傳回結果便會發出下一個請求，如此不斷循環直到所有工作皆被做完且傳回，如圖 9。因此從屬行程速度越快，要求和執行的工作量也就越多，反之越慢則所做的工作量越少，所以也可以說這是一種簡單的動態負載平衡機制。

#### 3.1.2 通訊考量 (Communication)

我們的演算法，會有一個主控行程 (Master Control Process) 和數個從屬行程 (Slave

Process) 在不同電腦上，從屬行程會送訊息向主控行程要求區塊以及將做完的工作回送給主節點 (Master Node)，主控行程則需發送某一區塊的座標給從屬行程，過程中會發生多次通訊，通訊的多寡及時間長短是平行處理效能好壞的一重要關鍵，而通訊又和區塊大小 (Block Size) 有密切關係。

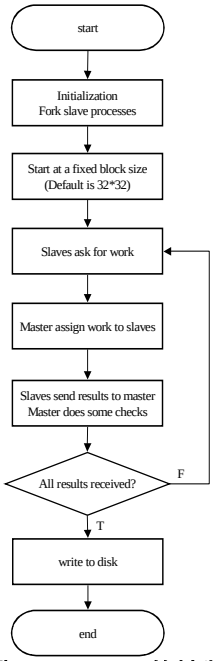


**圖 9 平行版 POVRAY 的控制流程**

如何決定區塊大小是一影響整體描繪 (Render) 速度的關鍵，區塊太大則每一 CPU 的負載量就大，發揮不出平行處理的效果，若區塊太小則 CPU 負載小，則經常要傳送分派工作的訊息或回傳描繪結果，造成過大的通訊額外負擔 (Overhead)。通訊額外負擔對於平行處理而言是一大致命傷，通訊越頻繁額外負擔就越大，甚至會導致用多個 CPU 的效能比單一 CPU 還慢。故原則上計算量 (Computation) 至少要大過通訊 (Communication) 時間，才會考慮傳訊息到其他機器由其他機器執行其他部分運算，此為決定顆粒度大小之基本原則。在找出顆粒度大小之底限前，必須先了解硬體通訊設備、通訊協定的延遲時間 (Latency)，程式流程中網路通訊的時間長短及時機。

以下就假設一解析度 320*240，描繪時間 2 秒，切割區塊大小 32*32 的圖像為例，說明通訊額外負擔發生點及時間長短：

(1) 首先若從屬行程 (Slave Process) 無任何工作要做，或已做完部分工作，則會向主控行程 (Master Control Process) 送出要求工作量的 PVM Message，此 Message 不到 1byte，約需 0.08ms。

(2) 主控行程收到從屬行程的要求訊息後，會送出目前尚未分派的 “Frame 編號、Block 編號、起始列、終止列、起始行、終止行”，六個整數型態資訊給要求工作的從屬行程，此六個資訊共 24bytes，依 PVM 之設計格式，傳送前必須包裝至一個 Buffer 內，所以包括包裝時間和實際傳送時間約共需 0.096ms。

(3) 從屬行程在接收到以上 24bytes 的分派資訊，會先解開包裝辨認資訊。在做完被分派的工作後回傳一個表示工作完成的 PVM Message 和計算結果，結果資料的大小需視區塊之大小而定，若以 32*32 的區塊而言，回傳資料大小為 20500 bytes，資料越大傳送時間就越久。

由上三步驟可知主控行程和從屬行程回傳一次工作量 (Workload) 的通訊時間為 0.08+0.096+ $T_{sl\_s}$，前二個值為定數，後面的 $T_{sl\_s}$ 代表從屬行程回傳計算結果之總時間，還需視區塊的大小而定。以解析度 320*240，2 秒可完成的圖，平均每一個像素需要描繪 2 / (320*240) = 0.026ms，經由網路延遲時間 (Network Latency) 測試結果表 (表 1) 可知，區塊大小至少要是 8*8，才能使計算時間大於通訊總時間。0.026*(8*8) = 1.664 > (0.08)+(0.096)+(0.08+0.313) = 0.569。

**表 1 交換器延遲時間**

| packet size(block size) | Lab switch(ms) |
|---|---|
| 24bytes | 0.096 |
| 40(1*1) | 0.092 |
| 100(2*2) | 0.105 |
| 340(4*4) | 0.139 |
| 1300(8*8) | 0.313 |
| 5140(16*16) | 0.684 |
| 20500(32*32) | 2.035 |
| 81940(64*64) | 7.633 |
| 327700(128*128) | 30.605 |
| 1310740(256*256) | 122.753 |

由以上例子若計算量要大於通訊額外時間，則區塊大小至少要是 8*8 以上的區塊。不過由於每張圖所包含的物件數量多寡和複雜度皆不一，加上節點 CPU 速度和網路速度等相關條件，都是訂定區塊大小時所要考慮的。

### 3.1.3 凝聚 (Agglomeration)

各從屬行程計算完之數值要需送回主節點 (Master Node) 彙整及寫入磁碟，但由於從屬節點 (Slave Node) 計算速度的不一致，因此可能較先分派的工作會晚傳回，後分派的早傳回，產生同步上的問題。我們的解決方式是讓主控行程等待全部的區塊都回傳完畢後再開始進行寫入磁碟的動作，如此雖會浪費較多記憶體空間，但在處理上會較為容易。此外由於受到某些過慢的從屬行程之影響，導致其他行程之計算結果皆已傳回呈現閒置狀態，主控行程仍持續等待過慢行程之結果，而使得整體時間延長和負載不平衡。於此，我們設計了再分派機制 (Reassign)。當所有的區塊都已被分派

完畢，而計算結果卻還未全部收到時，再分派機制就會發動，將還未收到的區塊再次分派給那些已經做完工作的行程，再分派仍是採需求導向 (Demand Driven) 的方式。另外在通訊最佳化方面，我們調整了一些 PVM 的參數，例如捨棄 PVM 所預設的 PVMd 對 PVMd 的傳送方式，改採 Task 對 Task 的傳送方式。

### 3.1.4 對應 (Mapping)

在行程的對應上，基本上可分為兩種。一種是一個從屬行程只對應到一個節點 (Physical Processor)，另一種是多個行程對應到單一節點 (Virtual Processor)，經實驗發現後者對應法在我們的環境下並不會得到較好的效能，故我們採用前者對應法。此外，我們在設計演算法和實作程式時，並非針對某一類型的叢集系統，對叢集系統的大小和處理器的種類速度也無限制，也就是所設計的平行程式可以在任一節點數量和速度的叢集系統上執行，也適用於異質硬體及作業系統平台。

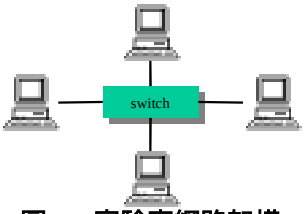


**圖 10 實驗室網路架構**

## 3.2 實驗結果

藉由以上的分析與設計，我們使用單台電腦，以及連接同一交換器的二至四台電腦所組成的叢集 (圖 10)，量測單機與電腦叢集之效能有何差距。首先很重要的是，在測量之前要把程式必循序執行之時間扣除，包括磁碟 I/O 時間、資料結構初始化 (Initialization) 和回收 (Destroy)、解析 (Parse)、列印資訊 (Print)、分出行程 (Fork) 等無法平行化的動作，而只計算程式中可以被平行化部分的時間。首先用單 CPU (PentiumIII-800 及 Pentium4-1.7G)電腦及原始 POVRAY 程式來執行圖檔，並得到一執行時間，之後再用二台 PIII、一台 PIII加 P4、三台 PIII、兩台 PIII加一台 P4、三台 PIII加一台 P4 電腦執行平行化的 POVRAY 版本，並量測執行時間。(電腦的主記憶體均為 256MB)

## 3.3 實驗結果與討論

在本研究中，我們選定五張圖：chess2、skyvase、desk、bucket、wg6 來評量平行化後的效能，圖 11 為以 PIII-800 基準量測每張圖的加速度。套用 Amdahl's Law 可求出三台 PIII-800 預期可增快 2.941 倍(3 / (1+(3-1)*0.01) = 2.941)，與實驗結果相近。由實驗數據並發現 P4-1.7G 約只比 PIII-800 快 1.5 倍，單純從 CPU 的時脈來說，1.7G Hz 應是 800MHZ 的 2.125 倍，這之間的差異是因為評估加速度不能只靠 CPU 之時脈，因影響加速度之原因還包括 CPU 架構及週邊系統架構等因素。由實驗結果總結發現，chess2 用解析度 800*600 的加速度最好；desk 用解析度 160*120 最差。

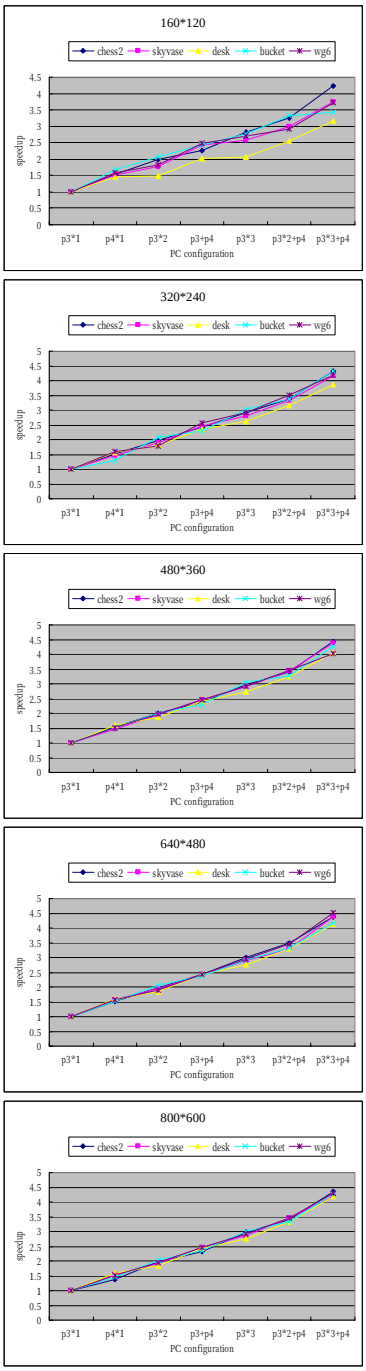


**圖 11 平行化之加速比較**

在本節我們所使用的需求導向法已算是種簡單的負載平衡方法，所以大體來說大部分情況下都還可獲得逼近線性的加速度，其中某些結果也頗符合 Amdahl's Law。不過以上實驗環境是在實驗室內用一個交換器連結所有的電腦，且 CPU 也只有 P4-1.7G 和 PIII-800 兩種，區塊大小似乎對加速度及負載平衡沒有很大的影響。但若網路環境或 CPU 速度的變異

增大，區塊大小的決定便變得相當重要且困難。例如當 CPU 速度差異大時，應縮小區塊以能應付各種速度之 CPU，改善負載平衡，但問題在一旦縮小區塊通訊時間便會增加，尤其當網路速度慢時，所增加的額外通訊時間更是可觀。所以，區塊大小的決定會影響到負載平衡和通訊時間，而負載平衡和通訊時間又是互相對立，所以需在何種狀況下使用何種區塊大小，是個相當重要但又難以抉擇的問題。

## 四、負載平衡改善

### 4.1 負載平衡問題

光跡追蹤平行演算法通常會先由主控行程 (Master Control Process) 依照叢集節點數量產生出數個從屬行程，每一個從屬節點都至少有一個從屬行程，之後主行程會依各節點的 CPU 速度來分派工作量，若 CPU 較快的節點會被分派到較大的區塊大小，反之 CPU 較慢就會被分派到較小的區塊大小，從屬行程並不會主動要求工作，所有的工作分派皆是由主行程以輪派 (Round Robin) 方式進行，當從屬行程收到指派工作後，會依序將工作放入佇列中，再從佇列中取出工作執行。此種演算法之負載平衡並不好，因為一開始要估計每個節點的速度以調配適當的區塊大小，若估計不正確就會使得區塊大小調配不當，也因此就會造成負載不平衡的情況。

在第三節所使用的需求導向法 (Demand Driven) 可大幅改善上述缺點，但如何制定適當的區塊大小 (Block Size) 仍是問題。在 3.1.2 節中，我們得知區塊大小需滿足計算時間大於通訊時間，但區塊大小之制定還需考慮到圖像複雜度、節點 CPU 速度、網路速度這三項相關因素。以圖像複雜度和節點 CPU 速度來說，圖像越複雜且節點 CPU 的速度不一致時，區塊就要縮小，以達較好的負載平衡，雖然會增多通訊次數及時間，但相對於大量的計算時間，通訊時間便顯得微不足道。不過，倘若節點 CPU 速度相當一致則縮小區塊只會徒增通訊時間；反之，若圖像簡單，則因計算時間短，通訊時間比重變大，就需減少通訊時間，故區塊要放大。但實際上在程式執行之前，我們只能知道圖像大概有幾個物件、幾條射線，卻難以精確得知圖像之複雜度和所需耗費時間，此外要評估節點 CPU 速度之一致程度也不容易。

以上討論是假設網路環境既快速又穩定，若各節點並非連接同一個交換器或在同一系統中，則區塊大小可能又需再次調整。因過慢的網路會使通訊時間增加，此時便要考慮將區塊縮小是否值得，也就是縮小區塊所帶來的負載平衡提昇後之效能是否高於其所浪費的通訊時間。綜合以上所述，制定區塊大小所要考慮的因素甚多，各因素間也都存有某些關係，而單純的需求導向法並無法自動制定出適合不同條件的區塊大小。

### 4.2 負載平衡改善方法

由於單純的需求導向法無法完全適應各種條件，所以我們將需求導向法做了些改善，設計出一個可以在程式執行中動態調整區塊大小的機制。使用者可在程式執行前隨意給定區塊大小，在程式執行時，若主控程式在等待時間內沒有收到來自某個從屬行程的訊息，且區塊大小仍在 4*4 以上。表示目前的區塊大小對那個從屬節點來說負擔太大，故將目前的區塊大小除以 2 後，再分派給它，否則仍照原大小進行分派。之所以要將區塊限制在 4*4 以上，是因經實驗發現 4*4 以下的區塊所造成的過大通訊時間，已遠超過所帶來的負載平增加後效益。藉由這樣的機制，自然會使得 CPU 速度快的節點的顆粒度大，速度慢者之顆粒度較小，每個節點依各自之能力而有不同的負載，如此便可改善負載平衡。

以上是節點間網路快的情況下所使用的方法，若在網路慢的情況，則會將等主控行程待時間延長，例如將網路快時主控行程 4 秒鐘等不到從屬行程之任何訊息，就將區塊大小除 2，延長至 5 秒鐘無接收到訊息再除 2，延長時間的長短要視網路速度之快慢。此為考慮到過慢的網路會使從屬行程要求或傳送工作的速度變慢，若仍依照原等待時間切割區塊則很可能會錯估遠端節點之計算能力，而切割出不適當的區塊大小。所以綜合以上，可將各情況下之策略以表 2 表示。其中快 CPU 是指足以應付某圖像之某種區塊大小的 CPU，慢 CPU 指無法應付某圖像之某種區塊大小的 CPU；快網路指的是連接同一交換器的區域網路，對應我們的實驗環境是指實驗室內，慢網路是跨越數個交換器的網路，對應我們的實驗環境是指實驗室到宿舍。較特殊的情況是在 CPU 和網路都慢時，若 CPU 比網路慢較多則區塊縮小，但若網路比 CPU 慢較多則區塊就增大。

**表 2 負載平衡改善策略**

| 網路 \ CPU | 快 | 慢 |
|---|---|---|
| 快 | 區塊大小不變 | 縮小區塊大小 |
| 慢 | 增大區塊大小 | 縮小區塊大小<br>增大區塊大小 |

藉由以上方式可以免除一開始時要制定區塊大小的困擾，程式會在執行時依圖像複雜程度及節點速度自動調整到適合的區塊大小，因 CPU 速度快和慢的節點之顆粒度並不相同，如此便可達到更好的負載平衡，此外由

於 CPU 快的節點之顆粒度不必隨著慢節點變小，故也可節省部分因不必要的小顆粒度所產生的額外通訊。另外主控行程的等待時間 (Waiting Time) 設定也是一項重點，若設太長則區塊可能會被切得過小，反之可能會太大。基本上我們可經由實驗找出適當的等待時間，首先找一個需花數分鐘以上描繪時間的圖，接著由短至長以每次增加一秒的方式依序給定等待時間，直到找出程式執行時間最短者，即為適合這個叢集系統的等待時間。

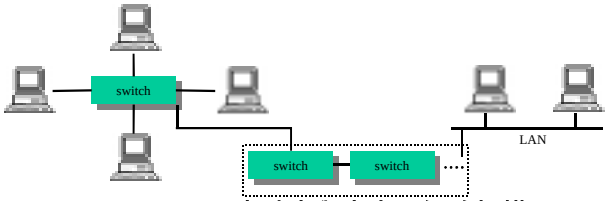


**圖 12 實驗室與宿舍間網路架構**

### 4.3 負載平衡改善之實驗結果

我們以實驗室中的一台 Pentium4-1.7G、三台 Pentium III -800 及宿舍中的一台 Duron-600、一台 Celeron-300，兩個不同的網段、共計六台電腦 (圖 12) 來評估負載平衡改善的程度，等待時間設定為六秒。由實驗數據發現，改善前後程式執行速度的提昇，以 chess2 的幅度最大(圖 13)，skyvase、desk、bucket、wg6 則幾乎與之前相同。究其原因，是其他四張圖都過於簡單，使各節點大都能在等待時間內計算完畢，區塊大小在程式執行中也就沒有任何改變，結果自然會與原程式無異；而 chess2 因圖像複雜度較高所以可以顯現我們負載平衡策略的優越性。

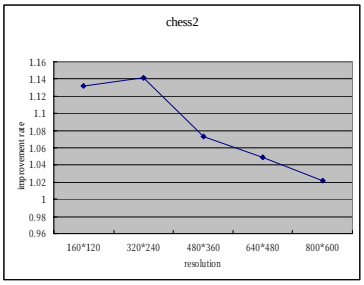


**圖 13 負載平衡改善前後加速比較 (chess2)**

### 4.4 討論

經負載平衡改善後，chess2 解析度由低至高分別比改善前增加了 13.2%、14.0%、7.3%、4.9%、2.2%，平均約 8.3%的效能。此實驗結果表示圖像越複雜，改善後增加的效能越多，也證實我們的改善方法確實可提升效能。主要原因是本研究的演算法，可以自動調整出適合節點 CPU 速度和圖像複雜度的區塊大小，故負載平衡可獲得提昇。不過由於我們的實驗平台仍算相當一致，叢集節點也不夠多，故似乎無法完全發揮本演算法之能力。若能有更複雜、更大規模的實驗環境，相信使用本研究的負載平衡改善法可更顯優越性。

## 五、結論

在本研究中，我們架設了 Linux 個人電腦叢集，在其上發展出一適合光跡追蹤法的行程場演算法，並以 PVM 來實作此演算法。此法是將圖像切成幾個區塊再由需求導向方式分派給若干個從屬行程執行，並有一主控行程 (Master Control Process) 做協調工作。此法的特點是 CPU 速度越快的從屬節點所做的工作會越多，慢節點做的工作會較少，所以基本上已達到一定程度的負載平衡。經二至四台電腦叢集的實驗結果顯示有近似線性的加速。

但隨著 CPU、網路速度的變異度增大，區塊大小的決定變得更趨困難，但上述方法並無法適應這些變異大的環境及幫助使用者決定區塊大小，也就浮現出負載平衡不佳的問題。故最後我們再設計出一改善負載平衡的方法，經負載平衡改善後，因主控行程可以自行動態調整出適合所有節點 CPU 速度和圖像複雜度的區塊大小，故負載平衡可獲得提昇。經實驗測量結果，發現改善方法平均上增加了約 8.3%的效能。

## 六、參考文獻


[1] G.M. Amdahl, “Validity of Sigle-Processor Approach to Achieving Large-Scale Computing Capability,” *Proc. AFIPS Conf.*, pp. 483-485, 1967.

[2] Beowulf at NASA CESDIS, http://www.beowulf.org.

[3] Beowulf Clusters at CACR, http://www.cacr.caltech.edu/beowulf/.

[4] CalTech Hyglac Beowulf Cluster, http:// www.cacr.caltech.edu/research/Beowulf.

[5] Ian Foster, *Designing and Building Parallel Programs*, Addison-Wesley, 1995.

[6] Highly-parallel Integrated Virtual Environment,http://newton.gsfc.nasa.gov/thehive/.

[7] Kai Hwang, and Zhiwei Xu, *Scalable Parallel Computing*, McGraw-Hill, 1998.

[8] Loki - Commodity Parallel Processing, http://loki-www.lanl.gov/.

[9] NASA - Goddard Space Flight Center, http://www.gsfc.nasa.gov/.

[10] POVRAY software, http://www.povray.org.

[11] PVM, http://www.epm.ornl.gov/pvm.

[12] Redhat Linux, http://www.redhat.com.

[13] UC Irvine Aeneas Supercomputer, http://aeneas.ps.uci.edu/aeneas/.

[14] XPVM, http:// www.netlib.org/ utk/icl/xpvm/xpvm.html.