一個高效率的擬亂數產生方法與平行化加速的研究與分析

鄭毓融[1] 林昀德[2] 林芳邦[2] 黃詠詳[2] 蕭一豪[2]
[1]資訊工業策進會雲端系統軟體研究所
thisisyujungcheng@gmail.com
[2]國家高速網路與計算中心
{ lsi,fplin,1203086,ihow }@narlabs.org.tw

摘要

亂數產生器可以產生一連串無法預測的數值，產生的數值是在有限的數值或有限的資料範圍內，產生出來的一連串數值通常具有分佈平均、隨機、獨立及不可預測的特性。良好的亂數產生器必須是長週期的，並具有良好的統計性質及有效率的電腦執行。本研究提出一種以亂數基底的計算方法來產生擬亂數值，節省的使用記憶體資源即可產生大量的擬亂數，並且產生的亂數不會重複，再導入多線程的平行化方法，有效利用多核心處理器的優勢來加速巨量的擬亂數產生。

關鍵詞：亂數，擬亂數產生器，平行化，多線程。

錯誤! 找不到參照來源。.關於亂數產生器與平行化的重要性

亂數產生器可以產生一連串無法預測的數值，產生的數值是在有限的數值或有限的資料範圍內。產生出來的一連串數值通常具有分佈平均、隨機、獨立及不可預測的特性，也就是每個數值出現的機率大致是相同的，且產生出來的數值無法由其他數值推導而得，每次所產生的數值之間沒有關係，即下一個數值無法由前一個數值推導出來[1][4]。

良好的亂數產生器必須是長週期的，並具有良好的統計性質及有效率的電腦執行。具有長週期的亂數產生器不容易發生重複性的亂數序列，經過很長一段時間後亂數才會再度循環；良好的統計性質則是指產生的亂數序列是具有獨立且連續分布均勻的性質；有效率的電腦執行是指電腦在使用相同的亂數種子下不論重覆演算幾次都會產生相同的亂數序列，此外演算法的速度必須夠快，執行的時間和占用的記憶體越少越好[2][6]。

各個科學研究領域中時常運用亂數產生器來進行各類應用的模擬與測試，例如：雜訊模擬、雜湊搜尋、抽樣、數值分析、決策等皆使用亂數產生器來實現。亂數產生的方法可分成物理性及軟體性兩種方法。物理性方法例如：擲骰子，擲硬幣和輪盤等是最簡單的產生方式；而利用更小的物質如原子或亞原子的不確定性物理現象也可用來產生物理亂數，雜訊、放射性、時鐘偏移等都是產生物裡亂數的來源。軟體性的亂數產生方法是運用特殊的演算法或人工方法來產生長週期性與隨機性的亂數序列，因為在軟體上所產生的亂數主要透過演算法所產生，並不是真實的隨機亂數，所以此類的亂數產生器一般也稱作為擬亂數產生器(Pseudo Random Number Generator)。常見的擬亂數產生器使用線性同餘法(Linear Congruential Method)、多階層餘法(Multiple Recursive Method)或梅森旋轉法(Mersenne Twister)等演算法來產生擬亂數[7]-[10]。

科學研究與實驗有時候需要大量的擬亂數來進行模擬，例如產生大量的二進位的亂數模擬檔案(符合0與1常態化分佈的Normal Distribution)或產生大量的正弦/餘弦波形來模擬一般常態的電子訊號。而在最近幾年很熱門的雲端計算環境裡面，實體機器與虛擬機器之間的關聯性與資源配置亦可使用亂數序列來做指定，如此一來除了可以加強避免容易被追蹤出來的安全性，也可以使資源分配更為平均。但是當我們所要產生的亂數序列數量越大且不能重覆及具備高維亂度時，其所需要的計算維度與執行時間也會跟著呈現指數般的遞增。

在高效能計算系統的發展中，由於半導體技術受到了物理限制的影響(包含：製程大小、散熱問題、電力消耗等)，使得電腦處理器的時脈速度面臨到瓶頸，所以處理器製造商轉以發展多核心的架構，在單一處理器中置入多顆運算核心來提升整體處理器的運算能力。多核心架構的發展提升了平行處理(Parallel Processing)的效能與優勢，所以近年來平行化運算因而被廣泛的使用與研究。

一般在單一台電腦上，將複雜計算進行平行化時可使用多線程(Multi-threading)的方式來實現；如果有多台電腦時，則可以使用訊息傳遞界面(Message Passing Interface, MPI) 的方式來實現。兩者都需要將原始程式進行改寫，通常我們會將計算與資料切割成多份較小的集合，然後分配到不同的

核心或各計算節點上進行運算，最後再將不同的核心或是各計算節點的結果收集起來做後續的處理，然後輸出最終的結果。本研究將使用多線程的方式來開發平行化版本的擬亂數產生器，並透過本研究所設計的亂數產生方法來分析產生大量擬亂數時的加速情形。

本研究所提出的亂數產生方法其執行效能比較快速同時也比較節省記憶體。但是當我們要產生非常巨量的亂數時，計算時間會隨著亂數數量呈現指數般的遞增，所以將會需要一段很長的計算時間。近年來，由於多核心技術越來越成熟，利用平行化的技巧可以加速超級巨量擬亂數產生器的執行速度，本研究我們是透過有效率地使用多核心處理器的計算資源來達到平行化的加速目標。

2.亂數序列的簡介與基本理論

亂數一般分成自然亂數與擬亂數。自然亂數基本上是無法逆推的真實亂數，也就是無法透過分析來推斷其下個或上個亂數值，一般來說，沒有人可以保證精準的猜測到下個亂數。一般只能透過物理性亂數產生器才能產生自然亂數。擬亂數則是透過人工或人為的演算法所產生出來的亂數，來模擬出自然亂數的隨機性，其亂數結果是根據種子來產生的，因此實際上是可以被計算出來的，不過透過使用越複雜的演算法與種子組合就可以讓人越難猜測到下個產生的擬亂數值，在應用上便可以達到自然亂數的特性[3]。

傳統產生擬亂數的方法通常會以電腦的實際時間(Timer Based)作為隨機種子並導入演算法來產生亂數，此方法所產生的亂數隨著時間單元的變化而產生不同的亂數，優點是快速且種子取得簡單，種子也不易重覆，足夠提供一般的亂數隨機性。然而此方法也有些缺點，例如：亂數的隨機性在高品質的亂數需求時，如加密的應用、統計或數值分析時就不適合；並且也可能會在數千萬次的亂數產生執行中遇到重覆的亂數。當透過以時間為種子的方式來產生不重覆的亂數時，擬亂數產生器需要提供一套重覆亂數的檢查機制，如果出現重複的亂數時，就必須捨棄此次產生的亂數，然後重新產生一個新的亂數，反覆執行直到獲得足夠且不重複的亂數數量為止。當以此方式來產生大量且不重覆的亂數，隨著已經產生的亂數數量越多，亂數重覆的機率就會越高，導致需要更多額外的亂數產生週期與亂數唯一性的比對時間，浪費大量的計算資源。

根據不同的需求及用途，擬亂數產生器可以被設計出產生多種不同類型的衍生亂數。例如產生 0 與 1 的亂數序列可用來模擬一般檔案的位元資料流；1 與 -1 的亂數可用來模擬傳輸訊號的震盪波形；包含文字與數字組合的亂數字串可以用來進行加密；或者是產生近似自然亂數序列來模擬出蒙地卡羅隨機取樣統計法[5]。

為了提供非常大量且亂數雜度足夠，同時具備不會重覆的擬亂數序列，本研究提出一種高效率的擬亂數生成方法來產生一連串的自然數亂數序列，此方法透過使用極少的記憶體來達到快速產生亂數序列的目的，並且保證產生的亂數序列數值絕對不會重覆。此外，我們把使用者輸入的密碼字串作為亂數隨機種子，以便確保輸入相同的密碼一定可以產生相同的亂數序列。最後我們再導入平行化架構，來提升整個超級巨量擬亂數計算的執行速度。

3.我們所設計的 PRNG 亂數生成演算法

相較於一般傳統的擬亂數產生器來產生不重覆的亂數序列，如圖 1 所示，本研究所提出的擬亂數產生方法是利用多組從 0 到 31 且不重覆的亂數序列作為多項式的計算基底(以 32 進制作為底數，如同一個二維陣列，內維大小固定為 32 個，外維大小則為階層的數量，至於外維的層數編號剛好就是所對應之指數次方的大小數值)，把每個階層的亂數值當作是係數(Coefficient)，然後乘上該係數所對應之 32 的指數率(次方值)，如此遞迴循環地相加各階層的數值組合而成，最後組合出 32 的 K 次方個亂數值。亂數序列的需求個數取決於欲產生的最大擬亂數值 N，然後以 32 為基底來進行多項式分解，找出可包含這個最大擬亂數值的階層個數 K。階層索引的次序編號也代表其各組亂數序列所需乘上 32 的指數次方值，如第一層的指數率為 0，也就是 32 的 0 次方；第二層指數率為 1，代表 32 的 1 次方；依此類推．．．。透過這種方式即可快速地計算出以 32 為基底且完全不會重複的 32 的 K 次方個亂數值，最後再輸出符合使用者所指定範圍內的亂數值序列。以下圖示說明了本研究的擬亂數生成原理與演算方法。

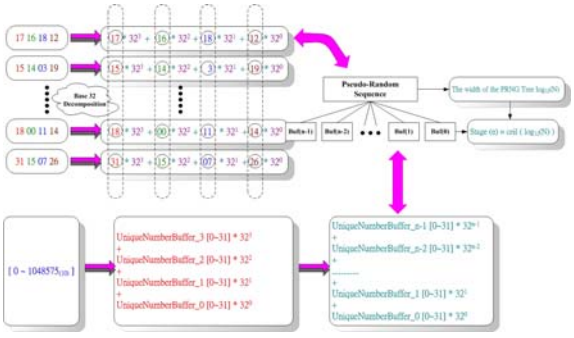

圖 1 以亂數基底方法產生亂數

如圖 2 所示，本研究的擬亂數產生器第一步先產生亂數基底序列。將使用者輸入的密碼字串做為亂數種子用來產生基底亂數序列。為了確保產生的亂數的雜度足夠，且每層的 Seed 種子數值能夠完全不同，首先將使用者輸入的密碼字串與預設的 128 個字元密碼字串做"替換"與"結合"的雙重組合，組合後的每個字元依序向前位移(Shifting)一個字元，使原本第一個字元循環左移至最後的字元，第二字元變為第一字元，最後再導入 MD5 Hashing Function 演算出第一層的 32 個 16 進制(Hex)的字元陣列(或稱字符陣列)。第二層依照第一層組合好的字串再進行一次字元循環左移，然後同樣再導入 MD5 Hashing Function 演算出第二層的 32 個 16 進制字元陣列，其他層也都依此方式來產生字元陣列，每層的字元陣列皆固定為 32 個字元長度。當產生所有需要的 16 進制的字元陣列(即陣列層數)後即可開始導出基底亂數序列。

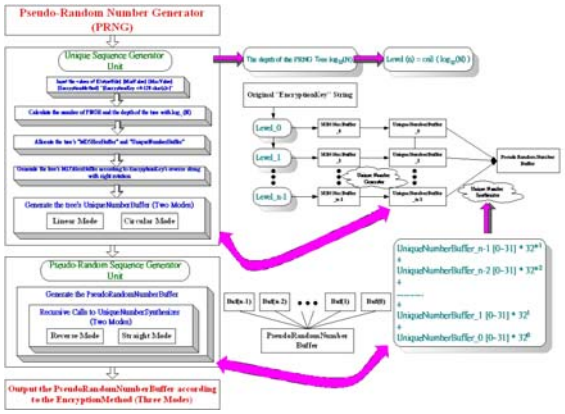

圖 2 本研究所設計之亂數產生流程

各層的字元陣列再透過本研究方法所設計的唯一數值(Unique Number)產生函數來轉換成 32 個 0 到 31 之間不重覆的亂數序列。由於數值的範圍不大，個數只有 32 個，因此可以快速比對是否存在重覆的數值，因此產生唯一數值的執行速度非常地快速。產生 MD5 基底亂數的方法是將字元以 16 進制先轉換成長整數(Long Integer)後做為起始位置，然後開始以起始位置(Start Position)的值加入計數值(Count)後與總字串長度(Key Length)相除取出餘數做為現在位置(Current Position)，現在位置帶入基底序列的 index 取出對應的位元，再做一次 16 進制的轉換，最後得到一長整數值(Value)。當最後得到的長整數值為偶數時，將長整數做放大處理，即是將字串長度減去長整數值再與字串長度相除取出餘數。透過上述的方法即可演算出 0 到 31 之間的數值，最後再檢查是否產生重覆的數值。當演算出 0 到 31 之間的亂數值發生重覆時，就進行雙向的鄰邊擴散演算(Bi-Direction Neighboring Diffusion)，搜尋最接近此數值可用的不重複數值。

雙向的鄰邊擴算演算以偏移(Offset)的方式進行向左或向右的擴散，也就是從周邊數值中取出尚未重覆的數值。當數值減去位移值後等於或大於 0 時或加上位移值後大餘總字串長度，則進行向左偏移擴散計算，反之則進行向右偏移擴散計算。透過這種持續的雙向偏移方式，檢查其值是否重覆，即可快速找出尚未重覆的數值。使用此雙向鄰邊擴散演算的優點是可以用最少的偏移(或搜尋)次數找出可用的不重複數值，無需再從頭開始檢查所有的數值(最壞的情況 O(n)=15，遠小於 O(n)=31)。

最後，當密碼字串轉換成計算亂數用的基底序列後即可開始產生 32 的 K 次方個擬亂數。當產生的亂數值越大時，所需的基底層數量則越多，以 32 的倍數來決定所需要的最大層數。例如當需要產生最小值為 1 至最大值為 100 萬的 100 萬個不重覆的擬亂數時，則至少需要 4 層基底，也就是至少產生 32 的 4 次方個亂數，再從中取出符合範圍內的所有亂數值。在進行每層的亂數值加乘組合時，每層的亂數值先乘以該層的 32 指數率後再開始做加法組合，層數越多，組合出來的亂數值就會越大。因為每層的亂數基底序列都沒有重複，所以將各層的亂數值相加後必然就可得到絕對不會重複的亂數序列。

利用本研究中所設計的擬亂數產生器所產生出來的亂數，也可以再轉換成 0 與 1 數量常態化分佈(Normal Distribution)的 0/1 亂數。所謂 0/1 亂數是指由 0 與 1 所組成的亂數，相當於計算機中的位元(Bit)0 與 1，所以可以用來模擬檔案資料或網路封包中的實際資料(Payload)。轉換為 0/1 亂數的方法有許多種，其中最簡單的方法是將亂數做餘數運算(Modulus)，將亂數值除以 2，即得到的餘數為 0 或 1，符合常態化均勻分佈的特性。

4.將亂數生成演算法進行平行化加速

亂數產生器產生亂數時，必須即時地儲存所有產生的亂數，一般都會使用陣列來暫存所有產生的亂數。使用陣列的方法需先定義一大小固定的陣列後，再開始將資料依序存入陣列中，資料不能儲存到陣列定義的大小範圍以外，否則就會發生溢位(Overflow)問題。本研究採用多線程的方式來實做平行化版本的擬亂數產生器，使用者能夠決定使用多少個線程數來執行平行化亂數產生，由於使用多線程後，每個線程所產生符合亂數需求範圍內的亂數數量將會不確定，無法有效地定義符合大小的陣列來暫存該線程所產生的亂數。本研究提出四種不同的方法來解決每個線程的亂數暫存問題，並比較四種方法的優缺點。

第一種方法是使用鏈結(Link List)的方法隨時依需求來增加或減少亂數暫存用的臨時空間。為了盡可能使用最少的記憶體，本研究設計了最簡單的結點(Node)資料型態，盡可能地讓結點資料型態使用最少的記憶體。結點型態中定義一個用來儲存亂數值的長整數(Long Integer)變數與一個用來指向下個結點位址的指標變數(Pointer)，透過該指標變數可實現單向鏈結(Single Link List)的方式來儲存亂數，每當一新的亂數值產生，則新增一鏈結點，用來儲存該亂數。使用單向鏈結的方式優點是能夠完全動態地依照產生的亂數個數來增加鏈結長度，但缺點是每個結點需要額外記錄下一個亂數結點的位址，一般來說在 32 位元的作業系統下，每個指標需使用 4 個位元組(Byte)來儲存記憶體位址，64 位元的作業系統則需 8 個位元組(實際大小依不同的作業系統與編譯器來做決定)。因此使用鏈結方法來儲存亂數值將會比使用陣列方法消耗更多的記憶體空間。

有介於鏈結的方法會使用較多的記憶體資源，因此使用陣列來暫存亂數序列，並透過 realloc()函數來動態調整該陣列的記憶體大小，即時地增加陣列的大小，避免產生溢位情形。其他三種方法分別使用了不同的 realloc()函數呼叫時機設計，第一種設計是每當產生一個新亂數時，便呼叫 realloc()函數，將陣列大小加一。第二種設計則是直接配置全部亂數數量大小的陣列，當亂數產生完畢後再呼叫 realloc()函數，來縮小陣列大小以符合實際產生的亂數數量。第三種設計是先將全部亂數數量除以線程數量後，得到一平均值 p(p=N/m)，各線程開始先配置 p 值大小的陣列，也就是將全部亂數數量大小的陣列先平均分配至每個線程中。當該線程產生的亂數數量超出預先配置好的陣列大小時，就開始使用第一種設計的方法，使用 realloc()函數來增加陣列大小；反之如果亂數產生完畢後，亂數數量低於陣列大小時，則使用第二種設計的方法透過 realloc()函數來縮小陣列，以符合實際亂數所使用到的陣列大小。

以上三種動態調整陣列的方法來暫存亂數值皆有其優點與缺點。第一種陣列方法的優點是可以完全動態的依照亂數產生數量來調整陣列大小，隨時保持最佳的記憶體使用率。然而，因為每產生一亂數時都需使用 realloc()函數來調整陣列大小，需頻繁地重新配置虛擬記憶體，因此會付出許多額外的時間在記憶體的配置上面，重新配置記憶體大小的執行次數大約等於亂數數量(N)。第二種方法的優點是使用 realloc()函數次數最少的，其次數大約等於多線程數量 m，但缺點是一開始就會佔用 m 個全部亂數陣列大小的虛擬記憶體(m*N)。第三種方法結合第一種與第二種方法的優點，一開始佔用的總虛擬記憶體空間為總亂數數量 N 的陣列大小(m*p=N)，至於 realloc()函數只有出現在亂數數量超出線程內的陣列大小時與最後在縮小陣列大小時才使用到，因此使用次數遠小於第一種方法。

本研究所設計的亂數產生器使用了多項式分解演算法，主要架構在多階層且長度固定的亂數基底陣列(Length = 32) 上面，因此在進行平行化時有多種方法可以將計算做拆解。最基本的方法為平行切割，將第一層的亂數計算基底進行切割。如圖 3 所示，例如當使用 4 個線程進行平行運算時，將第一層亂數基底陣列切割成 4 組亂數陣列(每組將會有 8 個亂數)，然後分別將各組亂跟底下各層(第 2 層至第 K 層)亂數基底陣列帶入各線程中進行運算。當各線程完成運算後，再按照順序從各線程中將產生的亂數序列組合後即可產生所有的擬亂數序列。不過使用這種切割方式的缺點是當使用的線程數量無法整除 32 時，則無法等量平均分配所有計算到各線程中；此外，由於每層只有 32 個亂數值，最多也只能使用 32 個線程。

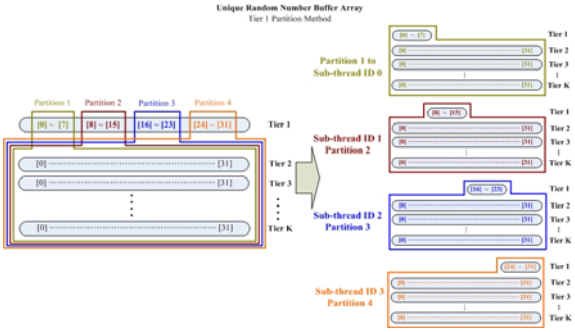

圖 3 第一層亂數基底序列切割

由於第一種平行化的計算切割方式有許多限制，因此本研究使用總量均分的方式來進行計算切割。進行總量均分切割時，直接將所有亂數計算基底陣列的組合進行切割，將切割出來的陣列範圍分配給各線程進行計算。所有亂數基底陣列組合相當於所有的亂數計算次數，如使用三層的亂數基底陣列時，亂數計算的總次數為 32 的 3 次方。總量均分切割後的各線程的計算範圍呈現一個垂直性的分割方式，以四個線程來說，第一個線程的陣列計算範為從[第一層的第 1 個/第二層的第 1 個/第三層的第 1 個]的開始計算，一直循序計算到[第一層的第 8 個/第二層的第 32 個/第三層的第 32 個] (相當於多維陣列[0][0][0] ~ [7][31][31])；第二個線程的計算範圍則從[第一層的第 9 個/第二層的第 1 個/第三層的第 1 個]開始計算到[第一層的第 16 個/第二層的第 32 個/第三層的第 32 個] (相當於多維陣列[8][0][0] ~ [15][31][31])。每個線程大致上都可以得到 8 * 32 * 32 個亂數計算組合，也就是大約產生的亂數個數。如圖 4 所示為上述之以總量均分的切割方法的意示圖。

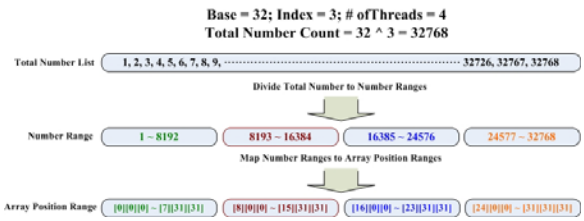

圖 4 總量均分的切割方法的意示圖

圖 5 所示為以總量均分方法的多線程詳細執行流程，根據所需的基底層數將亂數計算總量計算出來後，將總量以線程數量做均分，均分後每個線程得到在總量中的計算範圍，計算範圍再轉為陣列的中的開始與結束位置，即得到一陣列的計算範圍。每個線程再各自的陣列圍內計算出符合需求內的亂數後，將結果返回主線程，依序合併每個線程的亂數序列即可得到一完整亂數計算結果。

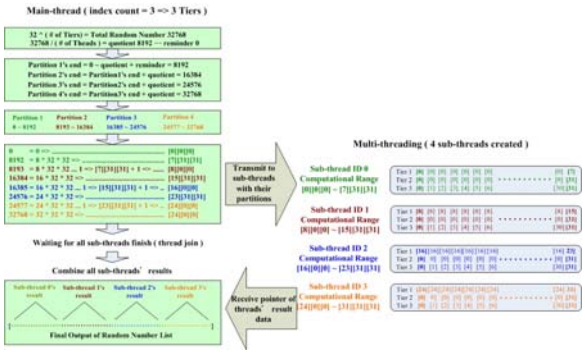

圖 5 總量均分的切割方法流程(4 線程)

當總量無法完全平均分配計算範圍到每個線程時(意即 m 無法完全整除 N)，我們將那些剩餘多出的計算分配給第一個建立的線程(通常線程 ID 為 0)。例如當所有亂數計算的次數為 32^4，使用線程數為 20 時，透過總量均分方式進行切割後，各線程平均分配到 52428 次計算，此時還有剩下的 16 次計算尚未分配，所以就將多出的 16 次計算加入到第一個產生的線程中，因此第一個線程會被分配到 52444 次的計算量，可轉換成 32 進位的 1 * 32^3 + 19 * 32^2 + 6 * 32^1 + 28 * 32^0 (相當於陣列組合從[0][0][0][0] ~ [1][18][5][27])。當總量均分無法完全平均切割時，多出來的計算次數一定會少於線程數量，所以把多出來的計算放入到任何一個線程對於整體執行效率而言幾乎沒有影響。但由於多線程的產生是依序執行的，一般來說當每個線程計算量一樣，越早產生的線程應該可以越早完成執行，所以將多出來的計算直接放入第一個建立的線程理論上是最有可能使全部的線程能夠同時完成計算的方法。

當使用總量均分切割方式來分割計算基底陣列除了可以有效地解決第一種分割方式中最多只能使用 32 個線程的限制；同時也能夠更均勻地分配亂數計算次數至各線程中，使平行化後的擬亂數產生器能夠更有效率地產生擬亂數序列。

5.平行化之實驗模型設計與實驗數據分析

針對本研究所設計的擬亂數產生方法與平行化的方式，我們從 1 個線程開始逐步增加線程數量（每次增加 4 個線程）來測試其擬亂數產生的加速效果，並針對四種亂數暫存方法測試其記憶體使用率。本實驗分成多線程加速測試及記憶體使用量測試兩個部分：(1).在多線程加速測試實驗中，分別以本研究中所設計的四種亂數暫存方法及亂數基底層數為 5 層、6 層及 7 層的亂數數量產生來做測試，記錄各層數的平均亂數產生時間。(2).在記憶體使用量測試中，使用 18 個線程，分別測試四種亂數暫存方法在產生約十億個(1G)擬亂數過程中的記憶體使用量，記憶體使用量的取樣間隔為 1 秒。表 1 所示為測試主機的硬體規格及實驗設計模型。

測試主機規格 **CPU:** Intel(R) Xeon(R) E7530 CPU @ 1.87GHz (true 24 cores CPU, HT enabled) **Memmory:** 529322584 KB **OS:** Linux 2.6.32-5-amd64 x86_64 GNU/Linux
測試準備 1. 快取清除 - "echo 3 > /proc/sys/vm/drop_caches" 2. 記憶體與硬碟資料同步 - "sync"
測試亂數數量 1. 多線程加速測試：產生擬亂數範圍從 1 開始到以下各層(index)的最大亂數值： ＊五層基底亂數{1048577, 2097152, 4194304, 8388608, 16777216, 33554432} ＊六層基底亂數{33554433, 67108864, 134217728, 268435456, 536870912, 1073741824} ＊七層基底亂數{1073741825, 2147483648, 4294967296} 2. 記憶體使用率測試：產生擬亂數範圍從 1 到 1073741824。
測試線程數量 1, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60
亂數暫存方法 1.Link List - Single Direction 2.Array Method 1 - Reallocate Array Size Random Number N Times 3.Array Method 2 - Reallocate Array Size T Threads Times 4.Array Method 3 - Reallocate Array Size 1 Time or size exceed N Times
記憶體使用率測試 1. 使用 ps -au 指令取得 VSZ 與 RSS 兩種記憶體使用量。 2. 測試 18 個線程數量的記憶體使用率。 3. 取樣間隔為 1 秒。 4. 從開始產生線程時開始記錄記憶體使用量直到所有線程完成計算。

表 1 測試主機硬體規格及實驗模型

本實驗之多線程加速測試部分，首先進行本研究所設計的原版擬亂數產生器(無多線程平行化之亂數產生測試。該擬亂數產生器為一般單線程程式，使用陣列來暫存所有產生的亂數，表 2 所示為其擬亂數產生時間之結果數據。相同層數(index)之亂數產生時間大致相同，最右側欄位為各層數的平均亂數產生時間。

Original PRNG Generation Time (seconds)				
Num of Index	**Num of Values**		**Generation Time**	**Avg. Generation Time**
index = 5	32 ^ 4 + 1	1048577	17.442	17.471
	32 ^ 4 * 2	2097152	16.973	
	32 ^ 4 * 4	4194304	17.375	
	32 ^ 4 * 8	8388608	17.670	
	32 ^ 4 * 16	16777216	17.809	
	32 ^ 5	33554432	17.559	
index = 6	32 ^ 5 + 1	33554433	698.639	703.980
	32 ^ 5 * 2	67108864	694.572	
	32 ^ 5 * 2	134217728	696.448	
	32 ^ 5 * 2	268435456	704.053	
	32 ^ 5 * 2	536870912	711.341	
	32 ^ 6	1073741824	718.831	
index = 7	32 ^ 6 + 1	1073741825	25948.942	26011.562
	32 ^ 6 * 2	2147483648	26010.398	
	32 ^ 6 * 4	4294967296	26075.346	

表 2 原版擬亂數產生器之結果數據

表 3 所示為本研究所設計的平行化版本擬亂數產生器的加速測試數據結果。其中左邊欄位為開啟的線程數量，由 1 個線程開始測試，逐次增加 4 個線程來測試直到 60 個線程。每種線程數量測試中也分別測試鏈結和三種使用陣列的亂數暫存方法。針對每種亂數基底層數（index = 5, 6, 7）內所有亂數測試的產生時間，加總計算取得該層的平均亂數產生時間。本實驗數據不包含亂數基底層數為 2、3 及 4，因為當亂數基底層數為 4 層及以下的亂數產生時間都遠低於 1 秒，所以無法有效地顯示出其加速效果。關於各基底層數的多線程加速曲線特性請參考表 4、5、6。

Num of Threads	Average Time of PRNG Generation (sec)											
	index = 5 {1048577, 2097152, 4194304, 8388608, 16777216, 33554432}				index = 6 {33554433, 67108864, 134217728, 268435456, 536870912, 1073741824}				index = 7 {1073741825, 2147483648, 4294967296}			
	Link List	Array Method			Link List	Array Method			Link List	Array Method		
		1	2	3		1	2	3		1	2	3
1	18.39	18.19	17.95	17.76	730.77	721.86	710.72	710.10	27411.18	27231.94	27206.41	27320.43
4	4.91	7.57	4.52	4.53	189.25	1525.99	177.73	177.40	6903.46	57735.83	6831.66	6843.92
8	3.02	4.98	2.30	2.30	106.20	1265.22	88.82	88.83	3482.71	10093.27	3419.55	3408.57
12	2.53	3.89	1.76	1.71	84.35	600.17	59.55	59.58	2356.13	16123.72	2279.62	2285.15
16	2.38	3.67	1.13	1.11	83.45	256.46	44.69	44.71	2027.33	12746.84	1786.12	1786.95
20	2.37	3.62	0.92	0.92	157.06	306.62	35.81	36.21	1653.19	7721.28	1431.27	1430.06
24	2.34	3.65	0.77	0.80	249.21	256.58	29.82	30.40	1414.60	11899.90	1199.95	1216.53
28	2.36	3.70	0.85	0.85	270.89	266.41	33.91	33.93	1420.65	6903.53	1366.39	1364.76
32	7.24	3.61	0.76	0.77	293.45	239.03	29.69	29.75	1417.06	6762.04	1202.81	1197.45
36	8.71	3.76	0.68	0.68	302.95	207.93	26.80	26.52	1475.52	6594.05	1073.29	1079.98
40	8.66	3.99	0.75	0.77	300.35	178.74	26.45	26.51	1739.71	5862.46	1108.14	1114.84
44	8.75	4.10	0.70	0.70	323.99	168.95	24.68	24.68	2150.40	5090.79	1035.14	1036.77
48	8.62	4.67	0.73	0.76	331.93	177.47	23.59	23.48	2267.76	5361.10	971.01	970.22
52	9.02	4.71	0.80	0.79	334.09	157.68	24.52	24.59	2657.33	4861.39	974.48	981.47
56	9.03	5.06	0.75	0.77	336.50	189.63	24.30	24.63	2756.16	3858.91	976.37	980.14
60	8.89	5.18	0.77	0.75	338.10	169.64	24.21	24.30	2832.96	4552.38	986.64	978.74

.表 3 多線程平行加速之實驗結果數據

表 4 所示為使用 5 層的亂數基底時，各線程數量的平均亂數產生時間曲線圖。以鏈結方法做為暫存亂數方法時，當使用線程數量為 24 時，會達到最佳的加速效果，但當線程數量為 28 時反而會開始降速，其後各線程數量的亂數產生時間約為 1 個線程時的一半時間（1 線程的平均亂數產生時間約 18 秒，36 至 60 線程的平均亂數產生時間約 9 seconds）。當以使用第一種陣列儲存方法時，多線程的加速效果大約在使用 12 個線程之後停止加速，之後開始緩慢降速。第二種和第三種陣列儲存方法比起之前兩種有更佳的多線程加速效果，但線程使用數量約達到 24 個之後就會停止加速。

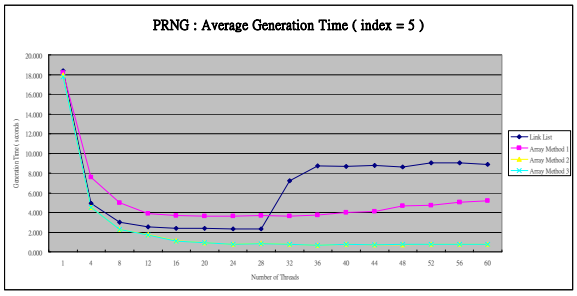

表 4 五層基底亂數平均亂數產生時間

表 5 所示為使用六層的亂數基底時，各線程數量的平均亂數產生時間曲線圖。鏈結的方法當線程數量為 16 達到最高的加速效果，當線程數量超過 16 之後加速效果開始緩慢下降。在使用第一種陣列儲存方法中，線程數量為 4 和 8 時，速度大幅下降，比 1 個線程約慢一倍左右，使用 16 個線程數量時加速效果最明顯，但是當線程數量大於 20 之後加速效果不明顯。使用第二種及第三種陣列儲存方法中，可以明顯看出有兩個階段的加速效果，當線程數量為 24 時得到第一階段最佳的加速效果，當使用 48 個線程時（第二階段）得到比第一階段更快的加速效果，之後加速效果趨近為零。

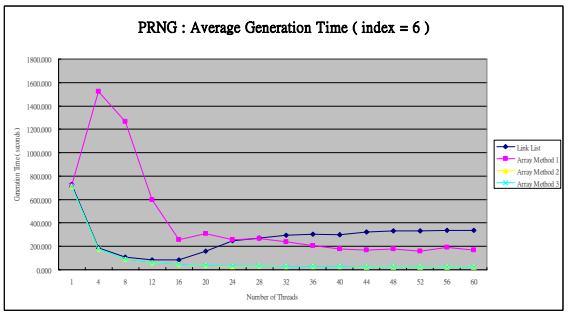

表 5 六層基底亂數平均亂數產生時間

表 6 所示為使用七層的亂數基底時，各線程數量的平均亂數產生時間曲線圖。當使用 40 個線程時，鏈結的方法達到最高的加速效果，之後開始緩慢降速。而使用第一種陣列儲存方法，加速效果是最低的，皆不如其他三種方法，使用四個線程時，加速效果為負的，當使用 8 個線程時開始逐漸有加速效果，但加速效果不穩定。使用第二種及第三種陣列儲存方法在第 24 個線程與 48 個線程數量可達到分別兩個階段的最佳加速效果。

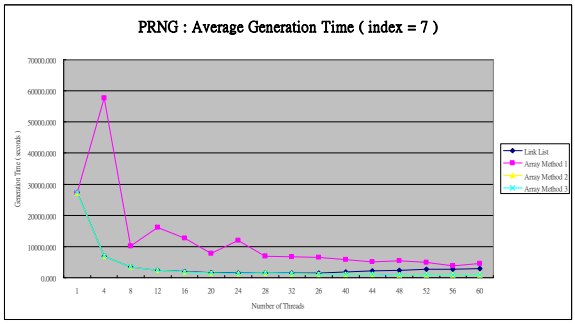

表 6 七層基底亂數平均亂數產生時間

將原始版擬亂數產生器與其平行化版本中之最短的亂數產生時間做比較，可得到平行化後的擬亂數產生器之最佳加速比率。表 7 顯示平行化版本的四種亂數暫存方法中，具有最佳加速效果的線程數量與加速比率。使用陣列儲存方法二和三，在六層亂數基底時，可得到 29.83 及 29.97 最佳的加速

比率。

		Link List	Array Method		
			1	2	3
index = 5	Num of Threads	24	32	36	36
	Best Speed Up Rate	7.46	4.83	25.35	25.50
index = 6	Num of Threads	16	52	48	48
	Best Speed Up Rate	8.43	4.46	29.83	29.97
index = 7	Num of Threads	24	56	48	48
	Best Speed Up Rate	18.38	6.74	26.78	26.80

表 7 最佳的多線程數量之加速比率

本研究所設計的四種亂數暫存方法解決了各線程無法配置準確的亂數暫存空間問題，使各線程能夠正常的暫存該線程所有產生的亂數。但根據測試結果，四種方法在不同數量的擬亂數產生，有些不同的加速情形，甚至降速。只使用 1 個線程時，四種方法的亂數產升速度差別不大，當增加線程數量後則開始產生差異。本研究中所使用的第二種與第三種陣列的亂數儲存方法的加速效果是幾近相同且最佳的。根據表 7，亂數基底數量為 5 時，第二種和第三種陣列儲存方法中使用 36 個線程時，最高可加速約 25 倍。亂數基底數量為 6 時，第二種和第三種陣列儲存方法中使用 48 個線程時，最高可加速至 29 倍 (近 30 倍)。亂數基底數量為 7 時，第二種和第三種陣列儲存方法中使用 48 個線程時，最高可加速 26 倍。另外可以明顯看到加速效果在 24 與 48 個線程時達到兩個最高的加速效果，這與我們所使用的測試主機規格有關，也符合了我們當初所預期的加速最佳理想值應當落在(a). 實際核心數量為 24 與(b). 超執行緒 (Hyper Threading)所模擬出來的 48 核心之間。

以下為第二部分的實驗測試，針對四種亂數暫存方法中，測試在產生大量亂數時的記憶體使用率。記憶體使用率測試使用外部系統指令(ps -au)來取得兩種記憶體使用量；(1). 虛擬記憶體(VSZ) 、 (2). 實際使用記憶體(RSS)。VSZ 為行程(Process)執行時，向作業系統所提出的記憶體使用量需求;RSS 則為行程執行時實際所佔用的記憶體使用量。

表 8 為使用鏈結時的 VSZ 與 RSS 使用率，兩種記憶體的使用率同時緩慢穩定成長。VSZ 最高使用約 34347 MB，RSS 最高使用約 33477 MB。

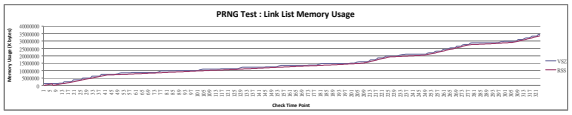

表 8 記憶體使用率之一(鏈結儲存)

表 9 為陣列儲存方法一的 VSZ 與 RSS 使用率。兩者記憶體使用率成反曲線成長，成長速度逐漸緩慢。VSZ 最高使用約 8647 MB，RSS 最高使用約 8360 MB。

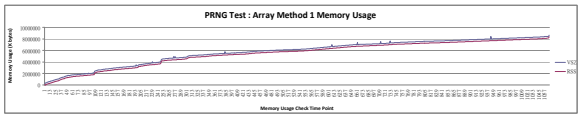

表 9 記憶體使用率之二(陣列儲存方法一)

表 10 為陣列儲存方法二的 VSZ 與 RSS 使用率。VSZ 全程固定使用大量的記憶體，RSS 的使用率慢速成長。VSZ 最高使用約 151300 MB，RSS 最高使用約 8329 MB。

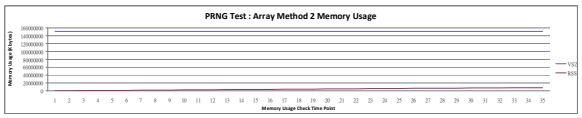

表 10 記憶體使用率之三(陣列儲存方法二)

表 11 為陣列儲存方法三的 VSZ 與 RSS 使用率。VSZ 全程的大小固定，RSS 則呈現線性成長。VSZ 最高使用約 8759 MB，RSS 最高使用約 8305 MB。

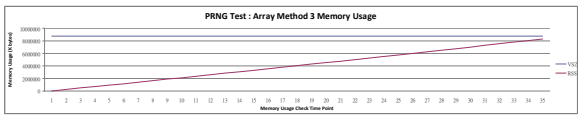

表 11 記憶體使用率之四(陣列儲存方法三)

四種亂數暫存方法中，使用鏈結的方法中 RSS 的使用量是最高，而使用第二種陣列儲存方法中 VSZ 的使用量是最高的，但這是由於使用多線程所導致的，因為每個線程自己都會配置一個全部亂數陣列大小的記憶體空間，因此使用量與線程數量成線性倍數成長。使用三種陣列的方法的 RSS 使用量都很相近，但測試結果中第三種的 RSS 使用量最低。

綜合多線程平行實驗的加速效果與記憶體使用率兩個部分的測試結果，使用第二及第三種陣列的儲存方法的加速效果最佳，而這兩者中，又以第三種的記憶體使用量較佳，因為 VSZ 的使用量遠小於第二種。換言之第二種陣列方法若是用在一些記憶體空間不大的主機中，很有可能會造成記憶體不足的窘境。

6.結論與未來工作

本研究所設計的擬亂數產生器可以有效地使用少量的記憶體來進行大量的擬亂數產生，確保產生的亂數絕對不會重覆，並且再透過多線程的平行化架構來提升擬亂數的產生速度，最高可達近 30

倍的加速效果。多線程以總量均分切割方式來將亂數計算更平均地分配至每個線程中，進而提升多線程的使用效率；同時也透過使用平均分配陣列大小再依實際使用量來調整陣列大小的亂數暫存方法提供了較佳的記憶體使用率。故我們所提出之擬亂數產生器具備了節省記憶體、執行效率快，可充分利用多線程平行化的方式來產生超大量的擬亂數等三大特色。

本研究已完成實作多層式亂數基底組合來產生大量擬亂數的方法與多線程版本的擬亂數產生器，未來我們可以更進一步地研究如何讓計算分佈更均勻的平行化切割方法與多線程之動態記憶體配置最佳化，最後也可以再導入 MPI 的平行化技術，透過網路串連多台電腦來計算更大量的擬亂數序列。

誌謝

[1] 資訊工業策進會雲端系統軟體研究所
This work is conducted under the "Cloud computing systems and software development project (2/3)"
of the Institute for Information Industry which is subsidized by the Ministry of Economy Affairs (MOEA) of the Republic of China.

[2] 國家實驗研究院國家高速網路與計算中心

參考文獻

[1] A. De Matteis, S. Pagnutti, Long-range correlations in linear and non-linear random number generators, Parallel Computing 14, 1990, pp. 207-210.

[2] Charles W. O'Donnell, G. Edward Suh, and Srinivas Devadas, "PUF-Based Random Number Generation", Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2004

[3] D.E. Knuth, The Art of Computer Programming, Vol. 2, Addison-Wesley, Reading, Mass., 2nd ed., 1981.

[4] David K. Gifford, "Natural Random Numbers", Laboratory For Computer Science, Massachusetts Institute of Technology. 1998

[5] H. Niederreiter, Random Number Generation and Quasi-Monte Carlo Methods, SIAM, Philadelphia, 1992.

[6] P. Hellekalek, "Good random number generators are (not so) easy to find", Mathematics and Computers in Simulation 46, 1998, pp. 485-505

[7] P. L'Ecuyer, Random number generation, In Jerry Banks (Ed.), Handbook on Simulation, Wiley, New York, 1997.

[8] 李和家、傅振華與陳樂惠，"強化型 LCG 虛擬亂數機制亂數產生評量之研究"，台灣博碩士論文知識加值系統，2010。

[9] 林水木與黃宇中，"真實亂數產生器之設計"，台灣博碩士論文知識加值系統，2003。

[10] 謝國偉與盧鴻興，"增量選取在平行化線性亂數產生器的應用"，台灣博碩士論文知識加值系統，2006。