Non-dedicated 叢集上 Loop Self-Scheduling 之研究

田尚修
靜宜大學資管系大四生
s9371048@pu.edu.tw

張沅合
靜宜大學資管系大四生
s9371006pu.edu.tw

王逸民
靜宜大學資管系教授
ymwang@pu.edu.tw

摘要

近年來，需要大規模計算及大量儲存空間的問題，越來越多，且日益複雜，為了幫助人們解決這些問題，科學家們努力不懈投入高性能計算環境的研究與開發，叢集系統就是其中一項重要的議題。另一方面，許多需要大量計算的程式常常包含許多平行迴圈，若能透過好的機制分配這些平行迴圈，在高性能計算環境上平行執行，必能幫助發揮這些計算環境的效能。

本論文提出新的方法，建議將 CPU、記憶體、message passing 所需的 communication cost 等因素同時加入平行迴圈排程演算法的設計，在 heterogeneous clusters 上發展更完整的平行迴圈演算法。另一方面，因為先前的研究大都假設在 dedicated clusters，也就是說，實驗時不受其他 processes 干擾，我們希望將環境延伸到 non-dedicated clusters，實驗時加入其他 processes 干擾，這樣會更接近真實的環境。我們在本系實驗室的 cluster system 上用 MPI 程式撰寫實際的平行程式，實作證實，當干擾發生時，原本在 dedicated 環境下效率好的方法，的確受到影響。另外初步實驗(僅考量 CPU)也證實我們的方法的確可行。

關鍵詞： non-dedicated 叢集、迴圈排程演算法、通訊負載

1. 簡介

近年來，需要大規模計算及大量儲存空間的問題，越來越多，且日益複雜，為了幫助人們解決這些問題，科學家們努力不懈投入高性能計算環境的研究與開發，叢集系統就是其中一項重要的議題。

另一方面，許多需要大量計算的程式常常包含許多平行迴圈，若能透過好的機制分配這些平行迴圈，在高性能計算環境上平行執行，必能幫助發揮這些計算環境的效能。這些機制，例如平行迴圈排程演算法，就是負責將這些平行迴圈，有效的分配給高性能計算環境上的各單位來執行，目的在縮短整個程式在此系統上的整體執行時間。

平行迴圈排程演算法基本上有靜態及動態二種。雖然這些方法在多處理機上已見成效，然而直接應用到 clusters 上則還有許多問題需要改良，例如 heterogeneous cluster 上各 nodes 的 resources 是相異的，分配方法不恰當，很容易產生 load imbalance 的情形，另外在 cluster 中 nodes 之間 message passing 所需的 communication cost，也是影響效能的主要因素。針對上述的問題，Chronopoulos 等人在 heterogeneous clusters 提出 distributed loop-scheduling scheme[2, 3, 4]，他們的方法主要精神是將 cluster 上各 node 的 run queue 上的 processes 數(等於是此 node 的 work load)，當作 master node 在 run time 時，分配工作給各 node 的參考，然而他們沒有考慮到 CPU 之外的差異，例如記憶體的差異。而目前最有名的是 Yang 等人在 heterogeneous PC clusters 上[1, 9, 10, 11]，提出二階段式策略：第一階段利用各 node 原始的硬體效能，採用 static 方式分配工作以避免 communication cost，第二階段則採用傳統的 dynamic scheduling algorithm 以 balance work load，然而他們的方法並未考慮到 non-dedicated cluster 環境下，也就是除了準備執行的程式之外，還有其他程式正在執行(有干擾的時候)，如果依然按照電腦原始的硬體能力去分配工作，那麼有可能會造成負載過重的電腦分配到過多的工作，造成叢集系統整體的 load imbalance。

我們首先設計一套受到干擾時的環境，測量當系統分享給其他 processes 時，效能是否會因為干擾的發生而降低，影響程度又是如何。實驗結果證實，當干擾發生時，原本在 dedicated 環境下效率好的方法，的確受到影響，因此有必要提出新方法加以改善。本論文提出新的方法，將實驗環境延伸到 non-dedicated clusters，這樣會更接近真實的環境。我們在本系實驗室的 cluster system 上用 MPI 程式撰寫實際的平行程式，以實作驗證其可行性。

本論文分為以下幾個部分：2.相關研究，

簡單介紹有關平行迴圈演算法的相關論文介紹；3.針對我們所提出來的方法作解說；4.實驗環境及實驗結果；5.結論，總結本研究的貢獻及未來發展方向。

2.相關論文介紹

平行迴圈（parallel loop）是平行程式在 High performance computing environment 上執行的來源之一[2, 3, 4, 6, 9, 10, 11]。平行迴圈排程演算法有靜態演算法及動態演算法二種。相關討論在先前許多論文已經有詳細的介紹，重要的文獻包含[5, 7, 8, 10, 11]，然而這些方法若直接 apply 到 clusters 上，會有下面的問題：傳統的方法大部分都假設各 nodes 上的 resources（CPU、memory 等軟硬體架構）都相等，而傳統的方法也未將 message passing 的 communication cost 考量加入。然而這種假設在 clusters 會有問題，因為 cluster 上各 nodes 是靠 message passing 交換資料，而且 cluster 上 nodes 的 resources 有可能是異質的。

針對上述的問題，Chronopoulos 等人首先提出在 heterogeneous clusters 上的平行迴圈排程演算法[2, 3, 4, 6]。他們建議的方法有下列二個特色：1.原來的各種演算法不用作太大的更動。2.將系統內各 nodes 上原始的 CPU power 及 run queue 內的負荷一起考量。在詳述此方法之前，首先簡述此方法需要符號如下：(其中 *i* 之值均介於 *1* 到 *P* 之間)

- 此 heterogeneous cluster 系統共有 *P* 個 nodes
- $Speed_i$ *代表 node i* 原始的 CPU clock 速度，$Speed_{min}$=1 表示速度最慢的 CPU
- $Queue_i$ 代表 *node i* 的 run queue 中 processes 之個數，即表示此 node 的 work load
- $A_i=Speed_i/Queue_i$，表示 *node i* 目前的 available computing power
- $A=$*所有* A_i 的總和，表示此 cluster 目前的 available computing power 總和

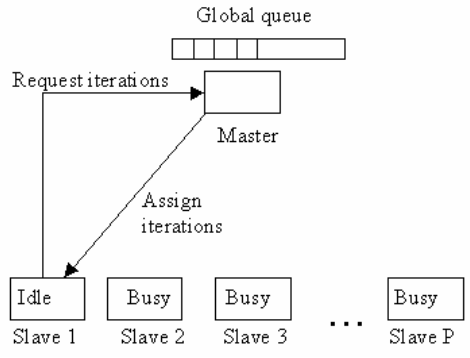

圖 1：Self-Scheduling scheme 的 system model

上面符號中以 A_i 最重要（$=Speed_i/Queue_i$），充分表現 *node i* 的 available computing power，其值越大，表示該 node 可以接受的 iterations 個數越多。圖 1 是最常見的動態 Self-Scheduling scheme 的 system model[10]，以這類動態 Self-Scheduling scheme 中有名的 **Factoring （FSS）排程演算法**[5]來說，Master 的 global queue 中存有所有 unscheduled iterations 的 indices，原始的 Factoring 方法如下：每一個 stage，master 會預先算好一整批的 *P* 個 subtasks，而且每個 subtask 所含的 iterations 個數都相等，subtask 的大小是當時 global queue 上剩餘的 iterations 總數(*R*)乘以 *1/(2P)*，*P* 是 CPUs 的個數。當某個 slave node 閒置時就向 master 要求一個 subtask 來執行。然而原來的 Factoring 方式是假設所有的 node 的 power 均相等，在 heterogeneous cluster 情形下就會有 load imbalance 的問題。因此 Chronopoulos 等人的方法建議依照每個 node 的 available computing power，將這一批 task 分配予閒置的 node 的 size 重新分配，分配的公式如下：

$$C_i = (R/2) * (A_i/A),$$

其中 C_i 是 *node i* 分配到的 iterations 個數。

Chronopoulos 等人的方法優點是依照各 nodes 在 run time 時 run queue 的 processes 個數加上 node 本身原始的 CPU 速度動態調整 assign 的 work load，然而卻無考量記憶體、網路因素等其他 resources 的影響。

另外 Yang 等人也在 heterogeneous clusters 上[1, 9, 10]，提出新的迴圈排程策略，他們的方法採二階段分配 workload 給 cluster 的 nodes：1.首先將 workload 的 α%，依照各個 nodes 的原始 CPU clock power 比例，利用靜態分配法分配工作，這一階段可以省去 communication cost；2.剩下的(100-α)%再用知名的 self-scheduling algorithms（例如 GSS [7]、TSS[8]及 Factoring[5]）動態分配，這一階段可以平衡工作負擔，至於 α 的值可以隨著實際環境調整。Yang 等人的方法因為同時兼顧 communication cost 的降低及 load balance，在 dedicated cluster 上表現非常好，很多最近的研究紛紛引用。缺點是除了沒有考慮到 CPU computation power 之外的差異，例如記憶體速度、記憶體容量、網路速度的差異，另外也未考慮到 non-dedicated cluster 環境下，當電腦本身還有其他程式正在執行，如果依然按照電腦

原始的硬體能力去分配工作，那麼有可能會造成負載過重的電腦分配到過多的工作，造成叢集系統整體的 load imbalance。

3. Non-dedicated Cluster 上新的 Scheduling 策略

Self-scheduling 演算法在 non-dedicated clusters 上仍然有改善的空間，因為我們不能只是依照每台電腦 CPU 原始運作能力及記憶體容量來分配工作，也要同時考量否有其他程式正在使用 CPU 或佔用記憶體所造成的影響。

本論文提出一個在 non-dedicated clusters 上，考量 CPU 原始能力及其使用率、memory 原始容量及其使用率，動態調整工作分配法則之 self-scheduling algorithm，希望依照這個法則可以提供叢集系統達到更好的資源使用率以及工作負載平衡。

在本方法中，master node 將動態得到每台 slave node 的 CPU 及 memory 使用率，利用這些資訊去計算出每台電腦在整個叢集系統中所佔有的效能比例，依照這個比例在加上好的 self-scheduling 方法，來分配工作，使得叢集系統達到更加的效率。

首先簡述我們的方法需要的符號如下[2, 3, 4, 6]：（其中 i 之值均介於 1 到 P 之間）

- P：叢集中電腦的個數。
- $Speed_i$：*node i* 的 CPU 速度，其中 $Speed_{min}$=1 表示速度最慢的 CPU，值越大表示速度越快。其實 CPU Clock 的數值並不等於 CPU 的能力，於是可以在實驗之前，讓每台電腦單獨做大量的運算後，以時間長短代表效能的高低值。
- $CPULoad_i$：*node i* 的 CPU 使用率，其中 $0 \leqq CPULoad_i \leqq 1$。
- $CPUPower_i$：*node i* 剩餘的 CPU 工作能力，$CPUPower_i = Speed_i * (1- CPULoad_i)$。
- ClusterCPUPower：用來表示叢集中所有的可用 CPU 運算能力合，由所有 $CPUPower_i$ 總和得來，其中 $1 \leqq i \leqq P$。
- $MemoryCapacity_i$：*node i* 的 memory capacity（$MemoryCapacity_{min}$=1 表示最小的 memory size；$MemoryCapacity_i$ 越大表示 memory capacity 越大）
- $MemoryLoad_i$：*node i* 的 memory 使用率，其中 $0 \leqq MemoryLoad_i \leqq 1$。
- $MemoryPower_i$：*node i* 剩餘的 memory 工作能力，$MemoryPower_i$ = MemoryCapacity$_i$ * (1-$MemoryLoad_i$)。
- *ClusterMemoryPower*：用來表示叢集中所有的可用 memory 能力合，由所有 $MemoryPower_i$ 總和得來，其中 $1 \leqq i \leqq P$。
- $FreePower_i$：代表修正後 *node i* 在叢集系統中相對的可用硬體能力，

  $FreePower_i$ = α*($CPUPower_i$ / *ClusterCPUPower*) + β*($MemoryPower_i$ / *ClusterMemoryPower*)

  其中 α 和 β 均介於 0 和 1 之間，且 $\alpha + \beta = 1$，α 和 β 用來表示 CPU 和 memory 對 algorithm 影響所佔的 weights。當 α=1 時，只依照 CPU 的影響分配工作;β=1 時，只依照 memory 的影響分配工作; $\alpha=\beta=1/2$，CPU 和 memory 的影響相同。本實驗初步僅考量 CPU 的影響(α=1，β=0)。
- *Threshold* 代表每次 master node 分配 iterations 給 node 時，iterations 數目的最低門檻值，其目的主要考量 cluster 上的 communication cost，每次分配的 iterations 數太少會增加 master 和 slave 之間 communication 的次數，也增加不必要的 communication cost。因此 *Threshold* 的值和實際 cluster 的 network speed 及 network 使用量有關，我們可以在執行程式開始時設定初始值，亦可在 run time 時依照實際情形改變。

我們所偵測到即時的系統負載資訊，可以準確的反映出系統在當時的忙碌狀態，而在結合系統原有硬體效能的資訊之後，便可以達到有效的叢集中電腦工作負載平衡。至於在取得系統負載資訊方面，可以使用系統本身提供的函式庫等工具，或是額外第三方的軟體。不管是使用什麼軟體，都可以很容易的將擷取到的資訊加入到傳統的工作分配法上面，而且在取得系統負載狀態所花費的時間是非常小的，在一併考慮進執行所需的工作的時間後，依舊可以達到比舊有方法更佳的效能結果。

我們可以將上述的方法應用在許多的工作分配法上，只要在每個分配工作的步驟上加入上述的規則，那麼就可以很簡單的將舊有的演算法改良為新的演算法。假設 master node 依照原始 algorithm，欲分配給 node i 的工作數量為 R_k，經過運算後修正後的工作數量就變成：

$$R_k * FreePower_i * P$$

上面符號中以 $FreePower_i$ 最重要，充分表現 *node i* 的 available Memory-CPU power，其值越大，表示可以接受的 iterations 個數越多。我們以 Factoring 排程演算法[5]來說明，我們

的方法和 Chronopoulos 等人的方法不同處，在於重新分配每個 subtask 的 iterations 個數的標準，除了 CPU 的 load 採用 CPU 使用率，而非 run queue 中 processes 之個數之外，乃依照每個 node 的 available Memory 加上 CPU power，而非僅考量 CPU power，方法如下：

Step 1：$C_i=[R/(2*P)]*FreePower_i*P$
$=(R/2)*FreePower_i$，其中 R 是目前共同列陣上的 unscheduled 的 iterations 總數。

Step 2：若 C_i 大於或等於 *Threshold*，則將 C_i 個 iterations 分給 *node i*，否則將 *Threshold* 個 iterations 分配給 *node i*，但若系統剩下的 iterations 數不足 *Threshold*，則僅 assign 最後剩餘的 iterations 給 *node i*。

Step 3: 將 R 的值減 Step 2 assign 出去的量。若還有未 assign 的 iterations 則等下一個 request，進行分配，一直到所有的 iterations 都分配執行完為止。

我們的方法也可以用來改良 Yang 提出的 α Self-Scheduling Scheme [1, 9, 10]，依照其兩階段分配工作法則：第一階段時我們不使用 CPU 的原始效能比例去作工作的分配，而是利用上面負載平衡的公式求出每台電腦開始工作時的可用工作效能比，來做第一階段靜態分配的依據；第二階段中，則是使用動態分配法加上 FreePower 來分配工作。各階段的工作分配數量可以由下列公式表示：

第一階段：$CS_i = FreePower_i *(N * \alpha\%)$

第二階段：$CD_i = C_i \times FreePower_i * P$

其中：

1. CS_i 為 node i 在第一階段所分配到的工作數量，N 為所有 iterations 數量，P 為 cluster 中 nodes 的總數。
2. C_i 為 node i 在第二階段中，原本利用 Self-Scheduling scheme 作動態分配所分配的工作數量，而 CD_i 則是修正後的結果。

4. 實驗環境與結果

4-1 實驗環境

我們利用現有電腦建構出一個異質性的叢集環境，利用這個環境下去實作我們提出來的演算法，並且利用一些程式實際去執行測量我們提出的方法是否在實行效率上優於舊有的方法。表 1 為我們叢集系統中電腦主要的配備，從表中可以看到叢集中某些電腦有著效能上非常大的差距，因此可以符合異質性叢集環境的條件。我們初部步不考慮記憶體的部份(也就是新方法中的參數 α=1、β=0)，所以表沒列出記憶體的規格。在這個環境之下，我們針對 Self-Scheduling Scheme，我們的方法加上 α Self-Scheduling Scheme，利用矩陣乘法程式來測量各種方法的執行效率。本實驗中，由於先前的經驗來看 TSS 可以擁有較好的實驗結果，所以我們在 Self-Scheduling Scheme 或是在 α Self- Scheduling Scheme 上，均只以 TSS 為實驗範本。

表 1：叢集系統中電腦主要的配備

節點	CPU CLOCK	作業系統
1	1.7G Hz	Fedora Core 5
2	1.8G Hz	Fedora Core 5
3	1.8G Hz	Fedora Core 5
4	3.0G Hz	Fedora Core 5
5	3.0G Hz	Fedora Core 5

實驗中，除了我們所提出的方法做每一次的工作量修正外，我們還針對了每台的 CPU 能力值作加權，這樣更能增加分配時負載的平衡。然而我們實驗的結果得知，CPU CLOCK 的數值並不能同等於 CPU 的能力，於是我們在實驗之前，讓每台電腦做大量的運算後，以時間最短代表效能最高。於是我們修正了表 1 的圖，用表 2 來表示：

表 2：叢集系統中電腦的能力值比

節點	CPU 能力值比	備註
1	1.0	能力最弱的值當 1
2	1.4	
3	1.4	
4	2.6	
5	2.6	

我們設計出了一套模擬環境受到干擾時的狀況，測量當 CPU 的使用率，因為干擾的發生而降低效能的程度。我們一共設計出了一些干擾程式，干擾程度的差異在於 CPU 使用率所佔之比例，輕度干擾佔了 CPU 使用率約介於 10%-30%之間；中度干擾介於 40%-60% 之間；高度干擾約介於 60%-80%之間。

4-2 實驗結果

在這一部份的實驗，我們採取的是 5000*5000 的矩陣乘法，分別針對不同個數的電腦分別測量程式執行的效率。最後我們使用

Self-Scheduling Scheme (TSS) 以及 α Self-Scheduling Scheme(改良 TSS) 來做為我們演算法改良的對象。

我們比較了不同程度干擾下，有考慮 CPU 使用率以及沒考慮的結果。由圖 2、3、4 可以看出來，我們所提出的方法在中、高度干擾的環境下，多台干擾的情況下，效能提升非常明顯，而輕度干擾的效能提升較少。

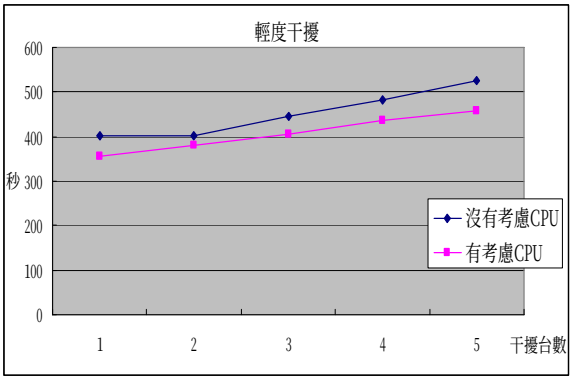

圖 2. 輕度干擾時，TSS 和改良 TSS 比較圖

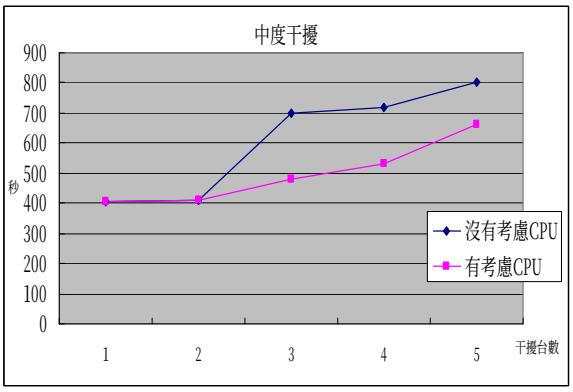

圖 3. 中度干擾時，TSS 和改良 TSS 比較圖

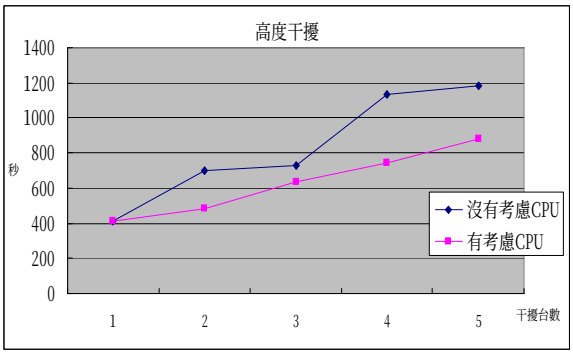

圖 4. 高度干擾時，TSS 和改良 TSS 比較圖

為了試驗我們方法對其他方法的影響，我們加入了楊朝棟教授所提出 α self-scheduling 的方法加以比較，並且實驗環境是在高度干擾下所進行，藉以觀察我們所提出的方法之效能。由表 3 可觀察到，單單考慮 CPU 效能的方式，在多台電腦時會有較好的效能，而αself-scheduling 的方式，有沒有加入 CPU 考慮似乎沒有影響，我們推估可能是因為 α 值太大，α=80，實際動態分配時的時間過短，由於我們的方法是在動態分配時加以調整，所以改善較為明顯。

表 3：高度干擾環境下的各種方法比較

電腦數	2	3	4	5
不考慮 CPU (原始 TSS)	4737	3256	1571	1157
考慮 CPU (改良 TSS)	4734	2975	1315	881
α self-scheduling(動態採用 TSS)α=80	4686	2999	1451	1034
考慮 CPU+α 方法(動態採用 TSS)α=80	4773	3035	1456	1032

在楊朝棟教授的方法下，較高的 α 值可以得到最好的效能，然而前提必須在沒有環境干擾的情況下。當環境干擾愈是嚴重，所造成的負載不平衡的狀況也就越嚴重，於是在加入我們的方法後，我們試著用較小的 α 值來測試是否有更好的效果；我們發現，在干擾越多台時，我們的方法可以保持穩定的結果，而當 α 值為 10 的時候，會有最好的結果，如表 4 顯示，無論干擾幾台的情況下，都比純粹考慮 CPU 使用率的方法好。

表 4 ：α 值高低對系統效能的影響

干擾台數	0	1	2	3	4	5
不考慮 CPU (TSS)	411	420	679	733	1122	1173
考慮 CPU (改良 TSS)	397	420	510	651	769	888
楊 α=10 考慮 CPU(動態採 TSS)	332	383	457	533	661	769
楊 α=80 考慮 CPUCPU(動態採 TSS)	330	714	713	1063	1089	1062

5. 結論

在本研究中，我們將異質性叢集環境中硬體的差異性以及各電腦中所存在的負載狀況加以綜合考慮，提出一個工作分配演算法則。在這個法則之下，我們實作了矩陣相乘程式，並且與舊有的工作分配方法作執行效率的比較，在各項實驗數據中都可以發現我們所提出方法的效能，在某些條件下可以優於舊有方法。我們提出的方法有另外一項優勢，那就是可以很容易的應用在舊有的演算法上面，因此

對現有系統架構不會有太大的影響。

影響一個叢集系統的效能因子中，除了電腦本身的效能之外，還有各電腦之間工作負載的平衡，特別是在異質性的叢集環境中。如果可以依照電腦的工作能力，很合理的將工作分配給各台電腦，那麼叢集系統就可以達到最大的效能。例如 α Self-Scheduling Scheme 就是一個非常好的方法，然而系統勢必會存在其他的程式在運行，因此只利用系統原有的效能來做工作分配，容易有工作不均衡的情形發生。

在本研究中，我們針對 CPU 的使用狀況，搭配硬體原本所擁有的處理能力作為工作分配的依據，從各種的實驗數據中可以看出，我們的想法可以有效的提昇系統的效能。

叢集中電腦的工作負載平衡於叢集系統是很重要的議題，本研究雖然針對其做相關研究，但其仍有許多課題尚待探討：

1. 在本研究中，我們只針對 CPU 的使用率作工作分配的修正，然而影響系統效能的因素卻不只有這個因素，記憶體、磁碟機 I/O 的速度，網路傳輸的快慢都有會造成系統運作效能的差異，因此在未來，可以對於這些還未考慮進的因素在做更加深入的研究與探討。
2. 干擾程式的修正、延伸。我們干擾程式的部份主要會影響的是 CPU 使用率的部份，然而未來我們還可以加入記憶體的考量。

致謝：

本研究的部分經費由靜宜大學專題研究計畫 PU96 11100 C25 及國科會研究計畫 NSC95-2221-E-126-005 提供。

參考文獻

1. K. W. Cheng, C. T. Yang, C. L. Lai, and S. C. Chang, A Parallel Loop Self-Scheduling on Grid Computing Environments, Proceedings of the 7th International Symposium on Parallel Architectures, Algorithms and Networks, 2004.
2. A. T. Chronopoulos and R. Andonie, A Class of Loop Self-Scheduling for Heterogeneous Clusters, Proceedings of the 2001 IEEE International Conference on Cluster Computing, pp.282-192, 2001.
3. A. T. Chronopoulos, S. Penmatsa, and N. Yu, Scalable Loop Self-Scheduling Schemes for Heterogeneous Clusters, Proceedings of the 2002 IEEE International Conference on Cluster Computing, pp.353-359, 2002.
4. A. T. Chronopoulos, S. Penmatsa, J. Xu, and S. Ali, Distributed Loop-Scheduling Scheme for Heterogeneous Computer Systems, Concurrency and Computation : Practice and Experience, 18, 771-785, 2006.
5. S. F. Hummel, E. Schonberg and L. E. Flynn, Factoring: A Practical and Robust Method for Scheduling Parallel Loops, Communications of the ACM Vol. 35 No. 8, pp.90-101, 1992.
6. S. Penmatsa, A. T. Chronopoulos, N. T. Karonis, and B. R. Toonen, Implementation of Distributed Loop Scheduling Schemes on the Teragrid, in Parallel and Distributed Processing Symposium, March 2007.
7. C. D. Polychronopoulos and D. J. Kuck, Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers, IEEE Trans. on Computers Vol.C-36 No.12, pp.1425-1439, 1987.
8. T. H. Tzen and L. M. Ni, Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers, IEEE Trans. on Parallel and Distributed Systems Vol.4 No.1, pp.87-98 , 1993.
9. C. T. Yang and S. C. Chang, A Parallel Loop Self-Scheduling on Extremely Heterogeneous PC Clusters, Journal of Information Science and Engineering 20, pp. 263-273, 2004.
10. C. T. Yang, K. W. Cheng, and K. C. Li, An Enhanced Parallel Loop Self-Scheduling Scheme for Cluster Environments, The Journal of Supercomputing, 34, 315-335, 2005.
11. C. T. Yang, K. W. Cheng, and W. C. Shih, On Development of an Efficient Parallel Loop Self-scheduling for Grid Computing Environments, Parallel Computing 33 467-487 2007.