行政院國家科學委員會補助專題研究計畫成果報告

分散式記憶體多處理機的自動工作量分配系統

計畫類別： 個別型計畫 整合型計畫

計畫編號：NSC 90-2213-E-002-145-

執行期間： 90 年 8 月 1 日至 91 年 7 月 31 日

計畫主持人：劉邦鋒

計畫參與人員：楊志學，沈子皓，陳麗如，許克仲，張力平．

本成果報告包括以下應繳交之附件：

赴國外出差或研習心得報告一份

赴大陸地區出差或研習心得報告一份

出席國際學術會議心得報告及發表之論文各一份

國際合作研究計畫國外研究報告書一份

執行單位：台灣大學資訊工程系

中 華 民 國 91 年 10 月 31 日

行政院國家科學委員會專題研究計畫成果報告

計畫編號：NSC 90-2213-E-002-145-

執行期限：90 年 8 月 1 日至 91 年 7 月 31 日

主持人：劉邦鋒 台灣大學資訊工程系

計畫參與人員： 楊志學，沈子皓 中正大學資訊工程系

一、中英文摘要

負載分配是平行處理中極爲重要的一個問題，在一個多處理機系統中，計算負載應該平均分配到所有的處理器來縮短總平行執行時間。如果負載不加以平均分配，會對平行效能提昇倍數產生負面影響，並無法充份利用多處理機系統中多處理機的計算效能。簡而言之，負載平均分配的目的是讓所有的處理器一直保持忙碌狀態。

在分散式記憶體多處理機系統中，除了將工作量平均分配外，資料的連續性(data locality)也是負載分配中極爲重要的問題。在此種系統中，處理器往往必需在執行工作上有先後順序，而需要同步，同時，由於分散式記體的關係，處理器之間傳遞資料必需經由訊息傳遞(message passing)，如果不保持資料的連續性通訊成本將會增加，因爲計算所需要的資料在大部時間由遠方經由訊息傳遞取得，因此，一個好的負載分配系統必需將工作量平均分配同時保証資料的連續性。

理想的負載分配困難的原因有下列三種理由。首先，計算以及資料結構可以展現出非常不規則的結構，導致我們無法在不影響資料連續性的情況下將計算量以及資料結構平均分配到各個處理器。其次，在某些動態問題中，各個資料的計算量會隨著時間而變化，所以負載必需做動態的調整。最後，在一般的非供平行計算專用的計算環境中，各個處理器所能提供的計算能力會隨著時間及外界因素而變化。這些外界因素往往無法在負載分配時能夠預測。

有許多負載分配方法採取主動的方式，無論是要切割一個迴圈，或是一個資料結構，或是工作時序圖(task graph)，這些負載分配方法都必需知道這些組成元件上面的工作量，才能做正確的負載分配，相形之下，一個被動的方法是將所有的工作放在一個共同區域，然後任何閒置處理器都能在這個共同區域取得工作來執行，雖然此種方法非常簡單，並且很容易在分享記體多處理器系統中來執行，但是它完全不考慮資料的連續性，所以在分散式系統中會造成極大的困難，除此之外，這個方法利用了一個全域的資料結構，所以需要一個處理器來執行集中管理，以保証資料結構的完整性。這會造成資料存取的集中性，以及較差的容錯性。

本計劃提出一個自動負載分配系統(WBRT)。此系統藉由工作偷取(work stealing)的方式來平均分配負載。工作偷取是由閒置的處理器到鄰近的處理器偷取工作來執行，所以並不需要知道各個組成元件的確實負載。換句話說，我們讓處理器，根據他們目前的負載自己去協調以達到平均分配工作的目的，也就是說工作並不是被分配到處理器，而是由處理器主動去爭取工作。

WBRT 也支援虛擬全域資料陣列(virtual global data array)，使用者程式可宣告一個全域資料陣列而由 WBRT 來負責負載分配及資料傳輸。使用程式可直接取用該陣列中的任何元素。除此之外，WBRT 也支援在資料分配邊界上的資料複製，同時讓使用者決定複製層次的大小，以上功能，完全由 WBRT 提供，使用者程式並不需要

知道任何有關實作的細節就可以使用。以交換技術爲基礎之路由方法已經成爲多處理機交換資訊的重要方法，但此種技術仍有其缺點。例如當一傳輸無法前進時，其使用中之網路頻寬及緩衝區接無法被其他傳輸所使用。爲了能有效解決此問題，同時能保持低傳輸時間及不規則網路所特有之可擴充性，本計畫提出一簡單網路架構—漸進式三角網格架構。此種網路架構可支援無死結路由及無衝突路由方法。首先我們證明在漸進式三角網格架構，任何最短路徑陸由方法皆不會產生死結，因此漸進式三角網格架構極適宜在調適性路由方法中擔任備用路徑。其次，我們也證明我們可以將漸進式三角網格架構中之處理器加以排序成一環狀順序，使兩個在此環狀順序中互不重疊的傳輸在網路中互不衝突。此種性質在無衝突路由方法的實做上有極大的幫助。我們初步的實驗結果也證實三角網格架構了所提供的路由方法比傳統的 up-down 路由方法能提供更好的傳輸效能。

關鍵字：負載分配，平行處理，資料的連續性，工作偷取

Abstract:

Load balancing is a very important issue in parallel processing. In a multiprocessor system the computation workload should be evenly distributed to minimize the total parallel execution time. Uneven distribution will degrade the overall parallel speedup since the aggregated CPU computing power is not fully utilized. In other words, we want to ensure that all the processors are busy all the time.

Besides even distribution of workload, data locality is another important issue in load balancing, especially for distributed memory multiprocessors. Most parallel applications require that processors synchronize and communicate with one another to acquire the necessary data for computation. If data locality is not preserved the communication cost will be excessive since most of the data will be fetched by expensive message passing. Therefore, a good load balancing system should partition the data evenly while preserving the data locality at the same time.

A good load-balancing scheme is difficult to find for the following reasons. First the computation and the data structure may be irregular. This non-uniform distribution of data and data access pattern makes it difficult to distribute data evenly while maintaining data locality. Secondly, load distribution must be dynamically adjusted, especially for those applications where the load on a data can vary from time to time. The computation must dynamically adjust the mapping in order to balance the workload throughout the computation. Thirdly, in a system that is not dedicated to parallel processing, the available computing power is constantly changing. Any sophisticated load distribution strategy could be easily defeated by an unexpected job submitted into the parallel system.

Many load-balancing schemes work in an “active” way. Be it partitioning the loop, or the data set, or the task graph, the load distributor must know the load information of all the elements it wants to partition (loop indices, the amount of computation on a data, or on a subtask) and make correct judgment accordingly. In contrast, a “passive” approach is to put all the work in a job queue and any idle processor can grab a job from this pool and execute it. This approach is conceptually simple, and can be easily implemented on a shared memory environment. However, this approach completely disregards data locality. In addition, the job pool approach uses a global data structure to store the jobs. The global queue requires a centralized control on the integrity of the data structure and will create "hot spots" in data access since every processor can only get job from a single processor. Finally, this would be difficult to implement in a distributed memory environment, in which processors can only receive remote data by message passing. We propose a passive scheduling system (WBRT) that distribute load by work stealing. To avoid having to know every details in active methods, we let the processor themselves to shift the load according to their current load. That is, jobs are not assigned to processor, but an idle processor will actively steal jobs from others. After all, the fundamental goal of load

balancing is to keep all the processor busy all the time.

WBRT also supports the idea of virtual global data array. The user program can declare a global data array and WBRT will partition it and balance the workload associated with it. The user application can access any data item in this data set. WBRT also keeps a “padding” around the boundary of the data set so that user applications do not have to explicitly keep track of this boundary data. This padding is initially constructed when the data structure is allocated, and incrementally updated by WBRT. User applications can use WBRT to access a virtual global data and do not need to know any details about WBRT.

Keywords: load balancing, parallel processing, data locality, work stealing

二、研究成果

WBRT system is a runtime environment for parallel programming on distributed networks. The main features of WBRT includes a high-level programming interface for array-based parallel programming, a partitioner and a WBRT scheduler for automatic data distribution and load balancing, and a communicator for data delivery and low-level message-passing over the network.

WBRT provides a global view of the data, in which the data structure is treated as a whole, with operators that manipulate individual elements and implicitly iterate over substructures. When a global array is instantiated, it creates a constituent local array on each processor. Whenever a kernel operator associated with global array is invoked, the operation is delegated to each local array. If communication is required, it is performed through the communicator. Conceptually, WBRT array operations are decomposed into parallel tasks, which are initially distributed to available processors following the “owner-computes-rules”. When the program starts execution, every processor self-schedules its own portion of the tasks, and when the need arises, tasks at processor boundaries are migrated among processors by work stealing technique.

Internally, WBRT implements an arrays based on the Single-Program-Multiple-Data (SPMD) model, in which every processor executes the same program operating on the portion of the array that are mapped to that processor. Since inter-processor communication may occur frequently during load balancing and actual program execution and the message start-up cost is usually high for network transmission, one-dimensional partition is adopted to minimize number of messages.

When WBRT initializes, it is given the size of the global array by the application. Then on each processor WBRT allocates a memory segment for this global data according to its current workload and memory usage, which results in a variable-size one dimensional block partitioning. After the global data structure is partitioned and mapped into local memory segments, each segment is partitioned into chunks consisting of a fixed number of adjacent elements of the global data. This chunk of data is called a task. The user code can retrieve a task from WBRT system, and perform operations in data-parallel fashion. The WBRT system provides data and the user provides the function that will be applied on the data.

WBRT provides a simple interface that the application programs can retrieve the tasks and execute them. The following code segment shows how a sample code communicates with WBRT runtime system. First the application calls a WBRT initialization function WBRTinit collectively. The application specifies the size of the global array, the number of data in

each task, the boundary width, and two function pointers that initializes and computes the data in a task (InitData and DoTask respectively). Finally the WBRTinit returns a WBRT handler by which the application can communicate with WBRT. Then the application calls WBRTRun to start the execution, and WBRTFinalize to finish.

A WBRT handler (WBRTH}) is the window that the application can communicates with the WBRT runtime system. The structure records detailed information of the runtime environment, including the geometry of the global array, the padding size, and the task size. Also it keeps a list of pointers to those tasks that have not been done, so that they can be exported to other processors for load balancing purposes. The following code segment shows the handler in details.

WBRT users can start the execution by calling WBRTRun, whose default implementation is also given in the following code segment. The WBRTRun function repeatedly calls WBRTGettask to get a task for execution. This task returned by the WBRT may be a local task or a remote one that was stolen from other processors. In other words, the task stealing is transparent to the application and it does not need to know where the task came from. All the details of sending/receiving data associated with the migrating tasks are handled by WBRT.

```
#include "WBRT.h"
#define DARRAYSIZE 500
#define BOUNDARY 1
#define TASKSIZE 5
int ARRAYSIZE = 20000;
typedef struct{
  int org[DARRAYSIZE];
  int res[DARRAYSIZE] ;
} DATA;
/* Major WBRT interface */
void WBRTInit(int argc, char **argv, int *arraysize, int tasksize, int boundaryprefetch, WBRTH* rh,
void(*INITDATA)(DATA*), void (*TASKFUNC)(Task*));
void WBRTRun(WBRTH *wrh);
void WBRTFinalize();
/* User functions to initialize and manipulate the data in a Task */
void DoTask(Task *);
void InitData(DATA*);
int main(int argc, char *argv[])
{
  WBRTH wrh;
  WBRTInit(argc, argv, &ARRAYSIZE, TASKSIZE, BOUNDARY, &wrh, InitData, DoTask);
  WBRTRun(&wrh);
  WBRTFinalize();
}
```