JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 18, 1037-1048 (2002)

Short Paper

Locality-Preserving Dynamic Load Balancing for Data-Parallel Applications on Distributed-Memory Multiprocessors

PANGFENG LIU, JAN-JAN WU* AND CHIH-HSUAE YANG+
Department of Computer Science and Information Engineering
National Taiwan University
Taipei, 106 Taiwan
**Institute of Information Science*
Academia Sinica
Taipei, 115 Taiwan
+*Department of Computer Science and Information Engineering*
National Chung Cheng University
Tainan, 701 Taiwan

Load balancing and data locality are the two most important factors affecting the performance of parallel programs running on distributed-memory multiprocessors. A good balancing scheme should evenly distribute the workload among the available processors, and locate the tasks close to their data to reduce communication and idle time. In this paper, we study the load balancing problem of data-parallel loops with predictable neighborhood data references. The loops are characterized by variable and unpredictable execution time due to dynamic external workload. Nevertheless the data referenced by each loop iteration exploits spatial locality of stencil references. We combine an initial static BLOCK scheduling and a dynamic scheduling based on work stealing. Data locality is preserved by careful restrictions on the tasks that can be migrated. Experimental results on a network of workstations are reported.

Keywords: load balancing, data locality, MPI, work stealing, data parallel computation

1. INTRODUCTION

Three factors affect the execution time of a parallel program on distributed-memory platforms: computation cost, communication overhead and delay. Computation cost is the time spent in the actual computation of the program and the delay can be caused by processors' waiting for messages or sitting idle waiting for other processors to finish. To reduce the idle and communication time, a parallel program must evenly distribute the workload among the available processors, and allocate the tasks close to their data. In other words, we need a load balancing scheme to handle this. The reason why we choose dynamic load balancing but not static scheduling can be summarized as follows:

Received September 12, 2001; accepted April 15, 2002.
Communicated by Jang-Ping Sheu, Makoto Takizawa and Myongsoon Park.

1. The amount of computation required for each task can vary tremendously between tasks and may change dynamically during program execution the work must be equally distributed among processors in a dynamic manner.
2. The data reference patterns may be irregular and dynamic. As they evolve, a good mapping must change adaptively in order to ensure good data locality.
3. The system load on the parallel machine may vary dynamically and is usually unpredictable. A change of a single processor load can easily defeat a sophisticated strategy for task (and data) assignment if we do not take this factor into consideration.

A good balancing scheme should address all these issues satisfactorily. However, these issues are not necessarily independent, attempts to balance the workload can affect locality, while attempts to improve locality may create imbalance of workload.

In this paper, we study the load balancing problem of data-parallel computations which exploit neighborhood data references. These computations are usually characterized by means of parallel loops characterized by iteration costs which are variable and often unpredictable, due to the dynamically changing external load. On the other hand, the data referenced by each loop iteration exploits spatial locality of stencil references, so that boundary elements can be identified once a partitioning strategy is given, and optimizations such as message vectorization and message aggregation can be applied. A large number of applications fall into this category.

Many load-balancing schemes work in an “active” way. They usually have a load balancer which actively balance the workload. But if the load balancer cannot acquire accurate load information, it could make a wrong decision. As a result the system load information must be constantly monitored and updated.

In this paper, we propose a passive scheduling system, WBRT, that achieves load balancing and, at the same time preserves data locality. We combine static scheduling and dynamic so scheduling that initially data are BLOCK distributed to preserve data locality for stencil-type data references, while dynamic load balancing is activated only when load imbalance occurs. Furthermore, to avoid the synchronization overhead required by a centralized dispatcher, we employ a fully distributed scheduling policy that constantly monitors and updates the system load information. Furthermore, to preserve data locality during program execution, migrations of tasks and data are in the way that preserves the BLOCK distribution as much as possible. Finally, we duplicate boundary elements (shadowing) between adjacent processors to avoid inter-processor communication for computation of boundary elements and also to improve vectorization of the loop body, hence reducing the computation time of each processor.

The rest of the paper is organized as follows. Section 2 reviews some related works. Section 3 outlines the implementations of the WBRT system. Section 4 reports our experimental results on two network of workstations, and section 5 concludes the paper.

2. RELATED WORK

2.1 Loop Scheduling for Load Balancing

Many studies have been carried out on various dynamic load balancing strategies for

distributed-memory parallel computing platforms. Rudolph and Polychronopoulos [1] implemented and evaluated shared-memory scheduling algorithms in the iPSC/2 hypercube multicomputer. It was not until early the 90's that load balancing algorithms for distributed-memory machines were reported in the literature [2-7]. Liu et. al. proposed a two-phase Safe Self-Scheduling (SSS) [2]. Distributed Saft Self-Scheduling (DSSS), a distributed version of SSS, is reported in [3]. DSSS is further generalized in [4]. Plata and Rivera [5] proposed a two-level scheme (SDD) in which static scheduling and dynamic scheduling overlap. A similar approach focusing on adaptive data placement for load balancing is reported in [8].

The difficulty of load balancing is in deciding whether work migration is beneficial or not. None of the above balancing strategies has addressed this issue however. The SUPPLE system [9] is a run-time support for parallel loops with regular stencil data references and non-uniform iteration costs. It is targeted for homogeneous, dedicated parallel machines and handles only load imbalance caused by non-uniform application programs. While the degree of this kind of imbalance changes only gradually, the load situations in a network of workstations may vary dramatically and frequently. Hence, the load balancing actions should quickly respond to the actual system load.

2.2 Dynamic Data Redistribution for Load Balancing

Another class of load balancing algorithms is based on changing the distribution of data periodically during program execution. A dynamic re-mapping of data parallel computations is described in [10]. It considers only data re-mapping when a computing phase change occurs. A similar work based on a heuristic approach was reported in [11]. The DataParallel-C group [12] formulates data-parallel computation by the concept of virtual processors. In [13], Feschet et. al. proposed a data structure called ParList for dynamic load balancing based on data redistribution for image-processing applications.

The potential problem with these data re-mapping methods for dynamic load balancing is that they under-utilize multiprocessor resources such as CPU, memory, and network, because the load-balancing is carried out in sequential phases that require global synchronizations. Our WBRT system employs a fully distributed, work stealing strategy to avoid global synchronizations.

2.3 Work Stealing for Load Balancing

The idea of work stealing is not new. Cilk [14, 15] is a multi-thread language with runtime support for dynamic load balancing. At runtime, an idle processor steals work from a *random* victim processor by migrating a task from the top of the job queue in the victim processor. On a shared memory environment, Cilk reported good speedup for various applications. However, the randomized work stealing strategy may perform poorly for data-parallel applications, where data locality is a critical factor in code performance. And Cilk does not make an attempt to analyze the profitability of work stealing either.

3. WBRT SYSTEM

In research reported here we implemented a workload balancing runtime system (WBRT) as a runtime environment for parallel programming on distributed networks.

3.1 Programming Model

Internally, WBRT implements an array based on the Single-Program-Multiple-Data (SPMD) model, in which every processor executes the same program operating on the portion of the array mapped to it. And a one-dimensional partitioning scheme is adopted to minimize the number of inter-processor communications.

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 sub-structures. When WBRT initialized, the global data structure is partitioned and mapped into local memory segments following the "owner-computes-rules". Conceptually, WBRT array operations are decomposed into parallel tasks. 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 (Fig. 1).

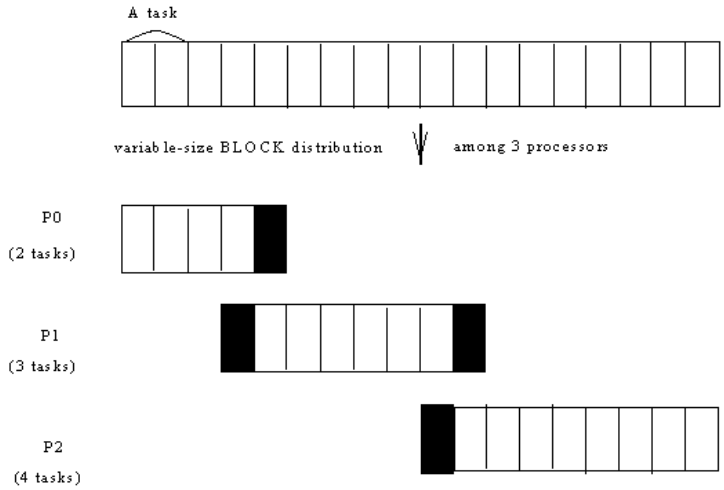

Fig. 1. A mapping of data array to processors. Each task computes two data elements and each computation on a data element references its two neighbor, so the padding size is one.

3.2 WBRT API

WBRT provides a simple interface for the application to operate on the tasks. Fig. 2 shows how a sample code communicates with WBRT runtime system.

A WBRT handler (`WBRT_H`) is the window through which the application can communicate with the WBRT runtime system. The structure records detailed information on the runtime environment.

WBRT applications start the execution by calling **WBRT_Run**. The **WBRT_Run** function repeatedly get a task for execution. The task returned by the WBRT may be a local or remote task that was stolen from other processors. In other words, the task stealing is transparent to the application, and the application 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 D_ARRAY_SIZE 500
#define BOUNDARY 1
#define TASK_SIZE  5

int ARRAY_SIZE = 20000;
typedef struct{
  int org[D_ARRAY_SIZE];
  int res[D_ARRAY_SIZE] ;
} DATA;

/* User functions to initialize and manipulate the data in a Task */
void DoTask(Task *);
void InitData(DATA*);

int main(int argc, char *argv[ ])
{
  WBRT_H wrh;
  WBRT_Init (argc, argv, &ARRAY_SIZE, TASK_SIZE, BOUNDARY, &wrh,
            InitData, DoTask);
  WBRT_Run(&wrh);
  WBRT_Finalize();
}
```

Fig. 2. A sample application using WBRT interface.

3.3 Implementation Details

A WBRT execution consists of two threads on each processor: The AP thread is the user application thread and the RT thread is the runtime system thread. An AP can request only tasks from the corresponding RT on the same processor. RTs work together to handle all the low level details of work stealing and task migration.

3.3.1 Initial tasks assignment

At the beginning of execution, WBRT distributes the workload according to the initial load on processors. First, each processor test-runs a task to determine its current load, then the processors distribute all the tasks among themselves accordingly. The

load information obtained in this way is most accurate. Formally, let the load on the *i*-th processor P_i be L_i, and N_all be the total number of tasks. The number of tasks given to the *i*-th processor P_i, denoted by N_i, is

$$N_i = N_{all} * \left(\frac{\frac{1}{L_i}}{\sum \frac{1}{L}} \right). \quad (1)$$

3.3.2 Work stealing

The most important function of WBRT is *work stealing*. When an AP requests work from a corresponding RT by `WBRT_Gettask`, the corresponding RT will give it a local task if one is available, otherwise the RT will try to steal a set of contiguous tasks from other processors. See Fig. 3 for an illustration.

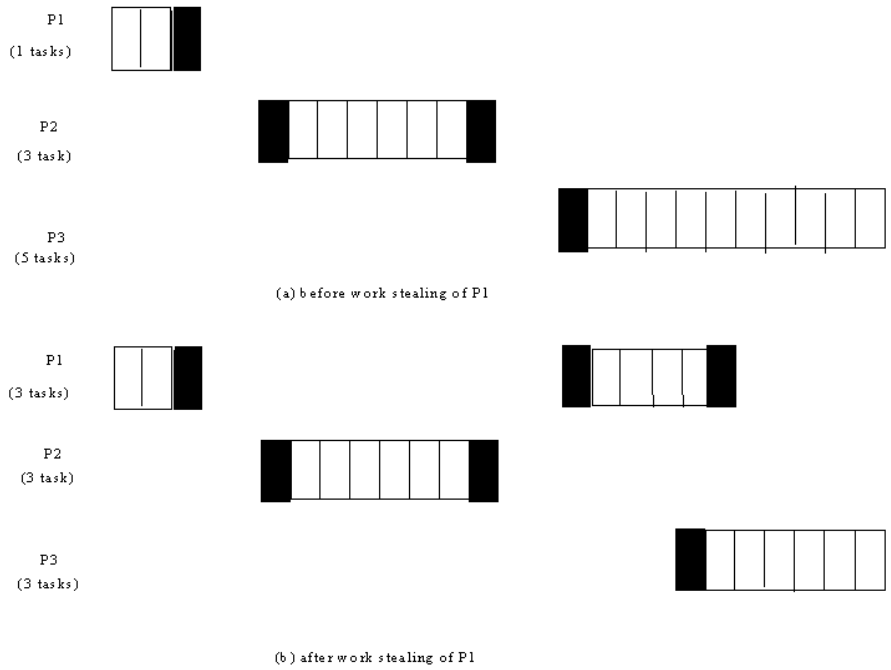

Fig. 3. A mapping of 9 tasks to three processors. (a) and (b) show the situation before and after processor 1 steals two from processor 3.

A processor must determine if it is *underloaded* before performing work stealing. Let S_i be the time for P_i to finish a task under the current workload, R_i be the number of remaining tasks in P_i. Then we define W_i to be the amount of time to P_i to finish its tasks.

$$W_i = R_i * S_i \quad (2)$$

We define that a processor P_i is underloaded if $W_i < k * W_{avg}$, where k is a constant that we can tune by experiments, and W_{avg} is the sum of all workloads.

An underloaded processor locates its victim for task stealing by message passing. It sends a *request* message to each of the other processors. A processor will return a *reply* message when it receives this request. And the requesting processor then determine if it wants to steal tasks from the replying processor.

A reply message from P_j consists of S_j, R_j, and T_j, namely, the current computation cost per task, the number of remaining tasks, and the time for P_j to send a task to other processors, respectively. The requesting processor P_i uses all this information to select the victim processor to steal tasks from.

To determine the most suitable victim processor we define a *cost ratio* for a possible victim processor P_j.

$$C_j = \frac{T_j}{W_j} \tag{3}$$

This ratio indicates the relative cost of stealing tasks from P_j among other choices. A processor with a small C_j is either overloaded or can send tasks to others very quickly; both indicate that it is a good candidate for work stealing. So we choose the one with the smallest C_j as a victim. Then we compare the victim's C_j with a fixed threshold δ. If C_j is smaller than δ, the requesting processor will steal from the victim. But if all the processors have C_j larger than δ, the requesting processors will *not* try to steal workload from others. We argue that under such a circumstance it will not be beneficial to migrate the tasks despite a load imbalance, since the communication overheads will be high.

After having located the victim processor, the underloaded processor will transfer tasks from the victim to itself. The victim makes sure the tasks that will be sent out form a *contiguous* block so that data locality is preserved, and only up to half of the tasks are allowed to be transferred. It will be impractical to maintain a single block partition at all times, since the most suitable victim processor for P, as far as cost C is concerned, may not be adjacent to P. To distribute workload and maintain data locality simultaneously, we make the following comprise that each processor can have up to a small number of contiguous segments. If the number of segments in a processor P reaches the limit, P's any further stealing must be adjacent to its existing segments. These restrictions reduce high communication costs in task migration and data fragmentation due to work stealing.

3.3.3 Boundary padding

WBRT maintains a read-only padding (shadowing) around the processor domain boundary. The size of this padding is set during `WBRT_init`. This padding is maintained by WBRT to simplify the application code. Each task being passed to `DoTask` is automatically padded by WBRT so that user code can directly access the data in the padding. In addition, when the boundary between two processors is changed, the padding is automatically adjusted by WBRT. The existence of padding is completely transparent to applications.

3.3.4 Synchronization

The scheduling between AP and RT is important. We implement AP and RT as two Pthreads in one processor. The RT will wake up every 500 microseconds to see if there is anything it needs to do. There is a tradeoff between shorter response time and better CPU utilization by AP in picking the length of the sleep.

3.3.5 Task execution order

At the beginning of `WBRT_Run`, every processor has a set of contiguous tasks. The execution order of these tasks may affect data locality, especially when work-stealing happens. WBRT uses a *middle-first* strategy, i.e., it chooses tasks from the middle of the local task set for execution, in order to keep data locality. When work stealing happens from either end, tasks may be given from the part nearest to the thief's local tasks. The purpose of this mechanism is to keep every processors' local tasks as contiguous as possible (Fig. 4).

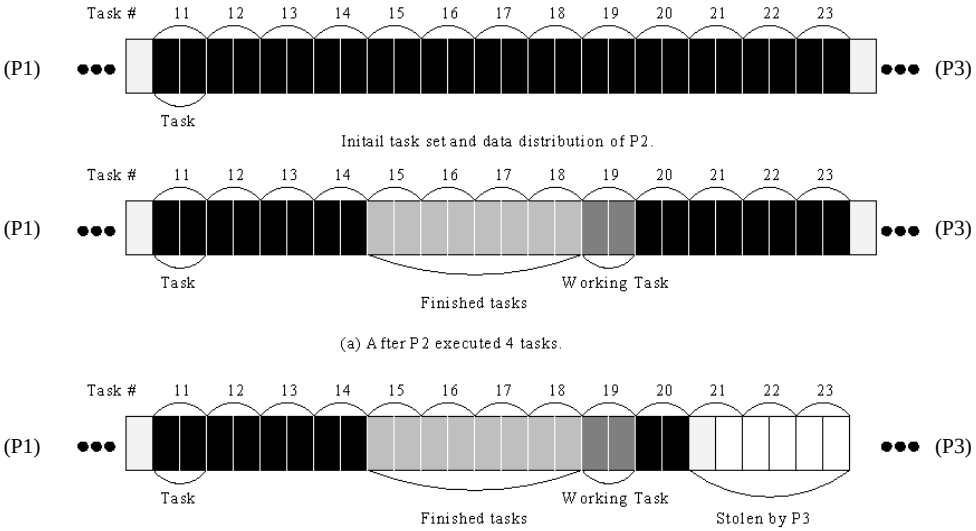

Fig. 4. An example showing task execution and migration. (a) indicates that P2 executed tasks 17, 16, 18, 15, and is executing 19 due to the middle-first strategy. (b) shows the configuration after P3 steals tasks 21, 22, and 23 from P2.

4. EXPERIMENTAL RESULTS

We design a series of experiments to evaluate the efficiency of WBRT on a cluster of four Sun Dual UltraSPARC II workstations. Each workstation is running SunOS release 5.6 on Dual UltraSPARC II 296Mhz with 1GB of memory, and we use mpich 1.1.2 for message passing, and POSIX thread for multi threading.

The application is a graphic relaxation process that computes the value of every pixel as the average of its four neighbors. The computation domain is an N by 500 matrix where N is between 1000 and 10000.

4.1 WBRT Runtime Overheads

First, we examine the overheads due to WBRT. We run the graphic relaxation code sequentially and compare the results with WBRT with/without workload stealing. Different problem sizes are tested. Fig. 5 shows that the speedup on the cluster of four Dual UltraSparc II workstations, with WBRT work stealing, is between 3.46 to 3.81. This substantial speedup indicates that WBRT API introduces only a very small amount of overhead.

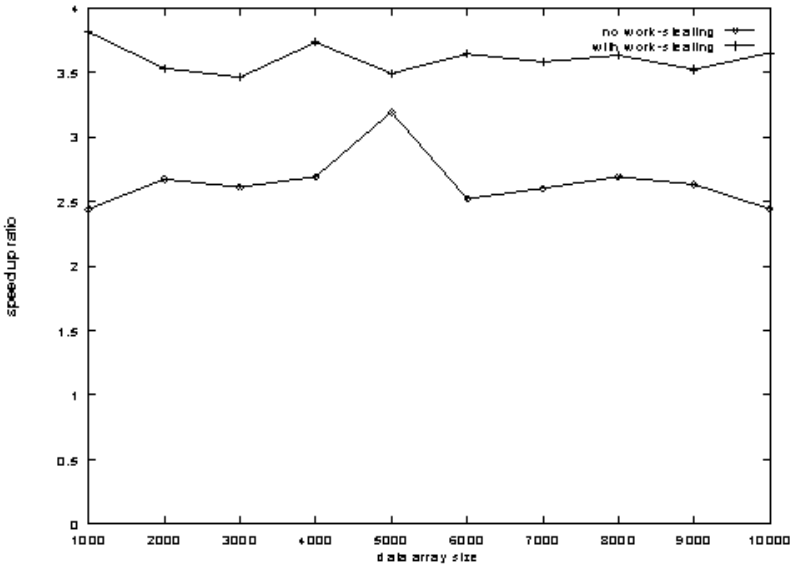

Fig. 5. Speedup of the relaxation code with and without work stealing the relaxation code sequentially, on a cluster of four Dual UltarSPARC II workstations.

4.2 Effectiveness in Load Balancing

The second set of experiments examine if work-stealing can effectively balance the load on a real cluster system. We run the same relaxation code on the UltraSPARC cluster, and compare the timing with and without work stealing. These two clusters are located at Academia Sinica and is heavily used from time to time.

On the cluster we run the relaxation code with WBRT. The experiments on this cluster show that the same code with work-stealing runs about 1.5 time as fast as without (Fig. 6). This significant improvement indicates that WBRT does reduce the parallel execution time on a multiprocessor with dynamic external workload.

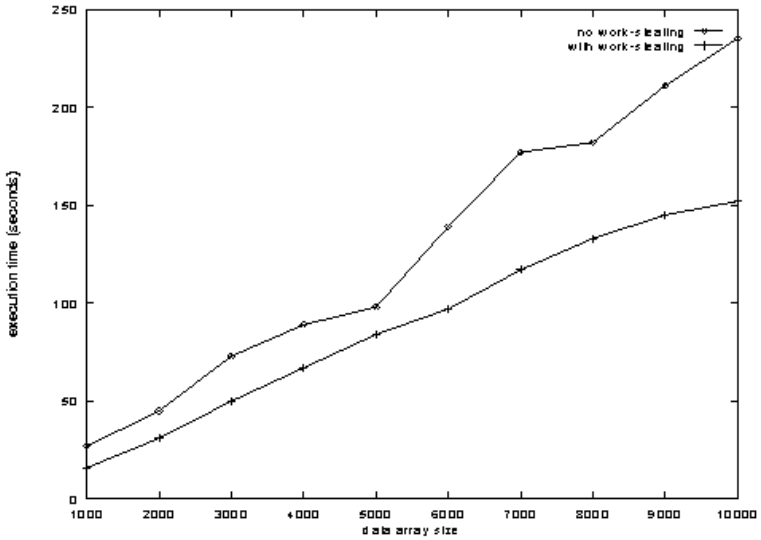

Fig. 6. Timing results from running the relaxation code on a cluster of four UltraSparc II with and without work-stealing respectively.

Fig. 6 also indicates that the execution time with work stealing increases more smoothly than without. In other words, WBRT with work that stealing gives much more predictable, and also shorter, execution time.

5. CONCLUSIONS

In this paper we show that the simple technique of work-stealing improves parallel efficiency. The key observation is that by letting the idle processors steal tasks from busy processors, every processor can be busy all the time. In other words, idle processors should *actively* search for tasks to execute, instead of letting a high level scheduler decide which task should go to which processor.

Data locality is important to parallel execution efficiency. By restricting that the tasks must be migrated as contiguous blocks, data locality can be preserved. This is as important as even distribution of workload, especially in a distributed memory multiprocessor.

Preliminary experiments show that WBRT work stealing effectively balances the load on a real cluster system. The speedups indicates that WBRT does reduce the parallel execution time on a multiprocessor environment with dynamically changing external workload. In addition, WBRT with work stealing also gives much predictable execution time than without.

REFERENCES

1. D. C. Rudolph and C. D. Polychronopoulos, "An efficient message-passing scheduler based on guided self-scheduling," *ACM International Conference on Supercomputing*, 1989, pp. 50-61.
2. J. Liu, V. A. Saletore, and T. G. Lewis, "Scheduling parallel loops with variable length iteration execution times on parallel computers," in *Proceedings of 5th IASTED-ISMM International Conference on Parallel and Distributed Computing and Systems*, 1992, pp. _____.
3. V. A. Saletore, J. Liu, and B. Y. Lam, "Scheduling non-uniform parallel loops on distributed memory machines," *IEEE International Conference on System Sciences*, 1993, pp. 516-525.
4. J. Liu and V. A. Saletore, "Self-scheduling on distributed-memory machines," *IEEE Supercomputing Conference*, 1993, pp. 814-823.
5. O. Plata and F. Rivera, "Combining static and dynamic scheduling on distributed-memory multiprocessors," in *the 1994 ACM International Conference on Supercomputing*, 1995, pp. 186-195.
6. M. Hamdi and C. -K. Lee, "Dynamic load balancing of data parallel applications on a distributed network," *ACM International Conference on Supercomputin*g, 1995, pp. 170-179.
7. T. Y. Lee, C. S. Raghavendra, and H. Sivaraman, "A practical scheduling scheme for non-uniform parallel loops on distributed-memory parallel computers," in *Proceedings of HICSS_____(全稱)-29*, 1996, pp. 243-250.
8. D. K. Lowenthal and G. R. Andrews, "Adaptive data placement for distributed-memory machines," *Internationl Parallel Processing Symposium (IPPS96)*, 1996, pp. _____.
9. S. Orlando and R. Perego, "A support for non-uniform parallel loops and its application to a Flame simulation code," *Irregular '97*, 1997, pp. 186-197.
10. D. M. Nicol and P. F. Reynolds, "Optimal dynamic remapping of data parallel computations," *IEEE Transactions on Computers*, Vol. 39, 1990, pp. 206-219.
11. A. N. Choudhary, B. Narahari, and R. Krishnamurti, "An efficient heuristic scheme for dynamic remapping of parallel computations," *Parallel Computing*, Vol. 19, 1993, pp. 621-632.
12. N. Nedeljkovic and M. J. Quinn, "Data-parallel programming on a network of heterogeneous workstations," *International Symposium in High Performance Distributed Computing*, 1992, pp. 28-36.
13. F. Feschet, S. Miguet, and L. Perroton, "Parlist: A parallel data structure for dynamic load balancing," *Journal of Parallel and Distributed Computin*g, Vol. 51, 1998, pp. 114-135.
14. R. Blumofe and C. Leiserson, "Scheduling multithreaded computations by work stealing," in *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. _____.
15. Supercomputing Technologies Group, MIT Laboratory for Computer Science, *Cilk 5.2 Reference Manual*, 1998.

Pangfeng Liu（劉邦鋒） received the B.S. degree in computer science from National Taiwan University in 1985, and the M.S. and Ph.D. degree in computer science from Yale University in 1990 and 1994 respectively. He is now an associate professor in the Department of Computer Science and Information Engineering of National Taiwan University. His primary research interests include parallel and distributed computing, design and analysis of algorithms, including randomized algorithms, routing and network topology, scheduling optimization, file comparison and FLASH file system for embedded systems. He is a member of ACM and the IEEE Computer Society.

Jan-Jan Wu (吳眞貞) received the B.S. degree and the M.S. degree in Computer Science from National Taiwan University in 1985 and 1987, respectively. She received the M.S. and Ph.D. in computer science from Yale University in 1991 and 1995 respectively. She was an Assistant Research Fellow in the Institute of Information Science, Academia Sinica from 1995 to 2000. She has been an Associate Research Fellow since Oct. 2000. Her research interests include parallel and distributed systems, cluster computing, and compilers and runtime support for parallel computing. She is a member of the ACM and the IEEE Computer Society

Chih-Hsuae Yang (楊志學) received the B.S. and M.S. degree in Computer Science, Department of Computer Science & Information Engineering at National Chung Cheng University in 1999 and 2001, respectively. Currently he is a research assistant of Institute of Information Science, Academia Sinica. His current research interests include parallel computing, parallel I/O, and grid computing.