A Migration-friendly Tasking Environment for Gardens

Ashley Beitz, Siu-Yuen Chan and Nickolas Kwiatkowski

School of Computing Science, Faculty of Information Technology,
Queensland University of Technology, Brisbane, 4000, Australia

Abstract. Gardens is a parallel computing environment that is capable of adapting at runtime to a changing set of available processors. It relies on the concept of migration to allow the application components (known as tasks) to move from one processor to another. This paper discusses the tasking environment needed to realise Gardens. The goal of this environment is to provide the runtime system with tasks that can easily and efficiently be migrated. It achieves this by employing the following features: a medium-grained unit of migration, global synchronisation, a heap management facility that supports migration and seed tasks.

1. Introduction

Gardens [1] is a parallel computing environment that supports efficient parallel computing across a network of workstations. The workstations are not dedicated to the execution of Gardens applications, so Gardens application components migrate at runtime to idle workstations to minimise the load on busy workstations. This strategy is known as adaptive parallelism.

Gardens application components are known as tasks. Tasks are supported by one of Gardens' sub-systems known as Gardens Application Tasking Environment (GATE). The goal of GATE is to provide the runtime system with a task that can be easily and efficiently migrated. GATE employs a number of features to achieve this, viz: a medium-grained unit of migration, global synchronisation, a heap management facility that supports migration and seed tasks.

The break-down for the rest of this paper is as follows: Section 2 will discuss GATE's model, Section 3 will discuss GATE's implementation, Section 4 will provide performance figures on migration, Section 5 will review related work, Section 6 will outline future work and Section 7 will present our conclusion.

2. The Model

Most existing approaches to migration use a process, an object or a thread as the unit of migration. A process is considered as a coarse-grained migration unit and is thus not suitable if frequent migrations are to take place. An object or a thread are at the other end of the spectrum - they are both considered as fine-grained migration units.

They are suitable if frequent migrations are to occur. However, using them as the unit of migration can increase the communication overhead between objects/threads, as you cannot make any compile-time assumptions about the sender and receiver being co-located in the same process. GATE is a kernel, which is based on a migration unit residing somewhere between the fine-grained and coarse-grained migration units. This medium-grained migration unit is suitable for frequent migration and clusters lightweight computational units (objects and threads) into a unit that will decrease the communication overheads between them. In addition to this, a number of other mechanisms have been added to the kernel to facilitate migration.

GATE's basic model is quite simple. In order for a piece of code to execute it has two fundamental needs - activity and resources. In GATE, a thread provides activity and a task provides resources. The thread is the unit of concurrency. In its simplest form it is a register context and a stack where call frames can be stored. Unlike a thread, which is an active object, a task is a passive one. A task is simply a collection of resources, which includes memory (a heap) and threads. A thread must always be associated with one and only one task. A task provides a thread with a context within which to execute. A process can support multiple tasks and a task can support multiple threads. The task is the unit of migration and when it relocates it takes its resources with it. Figure 1 illustrates GATE's basic model.

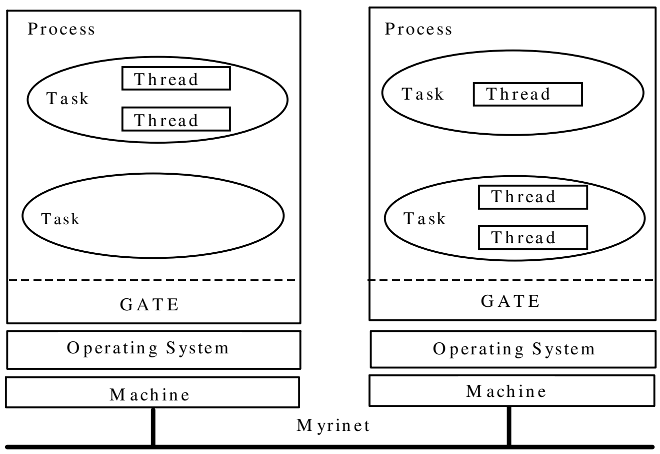

Figure 1: GATE's Basic Model

An asynchronous communication library called Active Messages [2] is used for inter-process communication. This library ensures efficient utilisation of the Myrinet [3] network, which interconnects every machine in the cluster. The asynchronous nature of Active Messages is achieved from two aspects: messages are delivered by invoking message-specific handlers and the host chooses when to poll for messages. The poll process delivers all messages in the message queue of the local host by invoking each message's handler.

The migration procedure needs to ensure that tasks will not try to communicate with other tasks during the migration process. This can be guaranteed by the use of global synchronisation. Global synchronisation is done at migration and ensures that all outstanding communications are processed before migration commences.

Semaphores have been provided to allow threads to synchronise with one another. Semaphores are an intra-task concept and should not be shared between tasks. When a semaphore is created it must be associated with a particular task. When a task migrates, it takes its semaphores with it.

The task state is in a form that will assist the migration mechanism. This is accomplished by storing everything that constitutes a task in the task's heap. This clusters all of a task's state into a single memory object, so migration simply requires for a task to move one memory object rather than multiple memory objects.

Each task‘s heap space can easily be moved from one process to another, because of the heap management model adopted. This model ensures that when a heap is moved to a new process that the same memory region used by the heap in the previous process is available in the new process. It guarantees this by globally reserving the memory region in every Gardens process.

Migration of active tasks can be a time-consuming procedure. Seed tasks have been provided to help alleviate this problem. Each seed task consists of the information needed to initiate a task with a thread of activity. A task in a seed state, ie. it has not yet started, can be more easily and quickly migrated than if it was active.

3. Implementation

3.1 Heap Management

The heap memory model used by GATE has been designed to provide low-overhead memory management and to simplify the task migration process.

Contemporary parallel systems that run on a cluster of workstations use either a distributed memory or shared memory model. Systems that use the shared memory model face the problem of cache coherence, as they continuously consume resources from the underlying runtime system. GATE does not have the cache coherence problem, as its heap memory model is based on the distributed memory model.

A general problem with task migration is handling pointers so that they still refer to the same data after the migration procedure. Our heap memory model avoids this problem by giving each task in the system a unique and disjoint heap address space. Thus, a task's heap uses the same address range throughout the task's life, regardless of where it migrates. To do this, a large chunk of virtual memory is allocated at system initialisation. This chunk of virtual memory is then partitioned into a numbers of mini-heaps. Since the Gardens system uses the SPMD parallel computation model across every participating workstation, a task’s address space will always remain the same. To ensure the mini-heaps used by the tasks are disjoint, a maximum number of nodes is defined at startup time and each node in the system is allowed to possess a different portion of mini-heaps from the system's entire set of mini-heaps. If one of the nodes is not participating, its set of free mini-heaps is not used by others and will be used again if it participates later. A mini-heap bitmap is used to keep track of the pool of free mini-heaps that can be used by the node. Every task created for a node is given a task heap from that node's free mini-heap pool.

The following example and Figure 2 illustrate this model in more detail. Suppose we have two nodes in the system, node A and B. After system initialisation, node A has the pool of mini-heaps in address space HA and node B has its pool in HB. Assume the parallel application creates four concurrent tasks, task 1 and task 2 on node A, and task 3 and task 4 on node B. Task 1, created on node A, uses the mini-heap HA0 and task 2, also created on node A, uses HA1. Similarly, task 3 and task 4, which are both created on node B, use the mini-heaps HB0 and HB1 respectively. Later, if task 2 is migrated to node B, its heap data will be sent to node B and stored into the address space HA1 of node B. If no more task migration occurs after task 2 has terminated, the mini-heap HA1 is freed and becomes a free mini-heap for node B.

The only heap management overhead during runtime is the updating of the local mini-heap bitmap. Other memory management activities like the allocation of space for objects are left to the compiler runtime system.

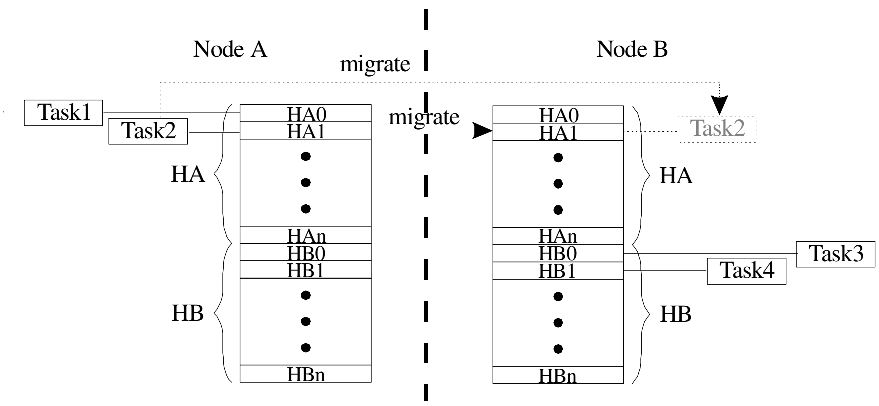

Figure 2: GATE's Heap Memory Model

3.2 Unit of Migration

Described earlier is the heap memory model used for GATE. This model supports migration by ensuring that each task has an identical address space on all nodes. Since there is no need to perform pointer translation, migration of a task's heap and any complex data structures contained within it requires only a block copy operation.

Everything that constitutes a task is included inside the task's heap, included are the task's descriptor, thread descriptors, semaphore descriptors and even the heap and stack descriptors used by the runtime system. This greatly simplifies the migration of a task and removes the need to extract these data structure from the kernel-level, as is done in some systems, since they are contained within the allocated region of the heap.

3.3 Global Synchronisation

Given the non pre-emptive nature of the computational model and that successful migration is required to maintain a consistent state for the task being migrated, migration can occur only at points where the task's state is stable. Within this model, this occurs when a task has chosen to communicate or block and allow another task to run. This leads to the idea that migration points and communication points are equivalent.

The communication issues of migration centres around messages sent to the migrating task. This is affected in part by the communication model being used, in this case Active Messages [2], with message handlers able to invoke replies. One solution would be the provision of a high-level protocol that would be added to provide on-forwarding of messages. However, the practice of on-forwarding is not preferred as it can lead to messages arriving out of order; necessitating, for example, the implementation of roll back mechanisms to handle those cases.

Global synchronisation flushes outstanding messages. The flushing operation is a pair-wise barrier operation, entailing the sending of a message to a host and waiting for the reply. This operation flushes all messages between the hosts. This is repeated for all pairs of hosts to globally flush all message queues.

3.4 Migration Process

Migration can only occur at a point where a task is prepared to yield execution. In this model, that point is when a task is ready to receive a communication. When a task blocks and a migration has been called, the kernel performs the following steps:

1. Inform hosts that a migration is about to take place by raising migration flags.
2. Globally synchronise and flush all the messages.
3. Perform migration on each task that needs to be migrated:
    - extract the task's threads from the source process's ready queue
    - transfer the task's heap to the destination
    - insert the task's ready threads into the destination process's ready queue.
4. Reset data structures, ie. migration flags and all copies of the task's remote address.
5. Signal completion of the migration operation.

With the global synchronisation and flushing completed, the tasks are migrated asynchronously using a bulk store operation and its respective handler. The asynchronous behaviour ensures no deadlock can occur from cycles in the task migration requests. Synchronisation completes the migration procedure with each host waiting for its reply indicating that migration is now complete.

3.5 Seed tasks

Seed tasks are a combination of a task constructor and a thread constructor, containing the information needed to initiate a task with a thread. A task in its seed state has not started and therefore represents a smaller unit to migrate. A seed consists of two parts: the code pointer and the shared object pointer.

The code pointer points to the procedure that will be called when activating the task's first thread. Since this is a SPMD model, this pointer has the same value on all hosts. The shared object pointer is a pointer to an object inside another task, typically the task that created the seed task. This pointer is used to perform remote method invocation and provides a mechanism for communication that is location independent.

A seed task in this model is very cheap to communicate as it is only two four-byte pointers. Seed tasks are not involved in any existing communication, so they can be migrated anytime without global synchronisation. These features makes them more suitable than active tasks for contemporary load balancing approaches.

4. Performance

The Gardens test bed consists of four 32Mb 120MHz Sun SPARCStation4's running Solaris 2.5.1. Each is connected to the Ethernet and the Myrinet networks. The underlying network that Active Messages will be using in this case is the Myrinet. The Myrinet is a low latency (approximately 10 μ seconds) 1.28 Gigabit/sec switching network with direct channels of communication between hosts. It is an error free network with no transport-level correction logic required, giving best effort message delivery with lower overheads.

4.1 System Measurements

The time taken to perform the global synchronisation and flush all outstanding messages depends on the algorithm and the number of messages. The average time taken is 68 μ seconds with a minimum time of 53 μ seconds (for four hosts). The important feature is the minimum time taken, which is the overhead for the synchronisation mechanism.

Time to migrate a task is also dependant upon the size of the task. Table 1 shows the total migration time for various task sizes. With regards to Table 1, the jump in the times between 4K and 8K occurs due to the overhead of packetisation, caused by the task size exceeding the 4K packet size of Active Messages.

Two main points are clear from the measurements taken for the migration system: the small amount of time needed to migrate a task considering the unit of migration and the relative constant time taken to perform the global synchronisation operations.

Table 1. Migration time for various task sizes

Size of Task (Kbytes)	4	8	16	32	64	100
Time for Migration (mSecs)	1.00	2.68	4.38	7.74	14.47	22.05
AM Transfer Time (mSecs)	0.46	1.60	3.21	6.41	12.83	20.03

4.2 Results of Task Migration - Comparison

Table 2 puts the migration times of various migration systems into context. The timing information was obtained from [4].

Table 2. Migration times for different systems

Runtime System	Time to migrate 100K unit (mSecs)
Sprite - SPARCStation1	330.0
Mach OMS - 33MHz Intel 80486	250.0
Choices - SPARCStation2	14.0
GATE - SPARCStation4	22.05

Each migration system uses a different migration algorithm:

- Sprite [5] uses the File Server algorithm
- Mach [6] uses the Demand Page algorithm
- Choices [4] uses the Freeze Free algorithm
- GATE uses the Total Copy algorithm.

GATE implements a total copy algorithm. As the name suggests, the algorithm totally copies the task and its environment before execution is resumed. The other systems use various algorithms to improve the time to perform migration. Each attempts to send the minimum information needed to resume the task or process. This leaves dependencies at the source host that have to be maintained for a undetermined period of time.

Comparing GATE to the other migration systems, the performance time seems to be much better. Especially considering that the algorithm used for GATE has no residual dependencies, unlike the other systems. The measurements presented for these systems do not include the overhead taken for the residual dependencies. The other aspect that is greatly different is the fact that all of the other systems use a 10 Mbit/s Ethernet, the latency and bandwidth of the Ethernet increases the time taken to perform the migration.

The main gain in performance achieved by GATE is through the use of the high speed network and pre-allocation of the heap structures associated with a task. In practice, the migration time on average is lower since tasks are sub-components of a larger problem.

5. Related Work

There are many groups working upon different approaches to the problem of migration. The four areas that distinguish GATE from the rest are: granularity of migration unit, migration of communication points, heap management and optimising the migration transfer. This section will cover the related work in each of these areas.

5.1 Granularity of Migration Unit

Parallel systems that support migration typically use an object, a thread or a process as the unit of migration. Those that use objects [7][8] and threads [9] are classified as having a fine-grained unit of migration. Those that use processes [10][11][5] are classified as having a coarse-grained unit of migration. A fine-grained unit of migration means that more configuration possibilities are available and that migration can happen very quickly, due to the small size of the unit being migrated. A coarse-grained unit of migration is best suited to a system that does not have much migration.

GATE's unit of migration - the task - is positioned somewhere in between these two classifications. Let us classify a task as a medium-grained unit of migration. The benefit of a task is that it is lightweight enough to support frequent migrations. At the same time it can support multiple lightweight computational units (threads and objects) and thus can support the close coupling of lightweight computational units with a high-level of cohesion, which should be co-located within the same process to optimise communication.

The only other system which provides a medium-grained unit of migration is UPVM [12]. Its unit of migration is called the user-level process (ULP) and it has similar properties to GATE's task. The main differences between UPVM's ULP and GATE's task are:

- A GATE task is a passive object, which may have active components. A ULP is an active object - it always has activity (a thread of execution) associated with it.
- Each ULP has only one thread of execution. A GATE task may have multiple threads of execution.
- ULPs can only be created when an application starts. GATE tasks can be created at any time during an application's execution and are thus more flexible.

5.2 Migration of Communication Points

When a task migrates, something must be done to deal with all of the communication requests that are sent to it during the migration procedure. There are two common strategies for dealing with this problem.

With the first, the receiver deals with the problem. It establishes a proxy in the task's place that forwards communication requests to the new address until it is asked to stop. An example of this strategy is used by DEMOS/MP [13].

With the second, the sender deals with the problem. When it detects a communication failure, perhaps due to migration, it looks for the new address and resends the message once it has successfully bound to the new address. An example of this strategy is used by the Distributed Computing Environment (DCE) with its automatic binding [14].

Global synchronisation, our strategy for dealing with this problem, is obviously very different from the other strategies. The other strategies deal with this problem by redirecting the communication channel, creating overheads at either end. Global synchronisation gets all tasks to a point where all generated communication has been processed. At this point, migration can take place without any concern about incoming communications. When migration completes, communications can resume without interference.

While this may seem draconian to perform large scale synchronisation, it should be noted that the mechanism is not intended for the migration of a small set of tasks. For the purpose of load balancing, migration of seed tasks is by far more efficient. The main purpose of such a mechanism is for the case where nodes become unavailable and the cost of synchronisation would be amortised over a larger set of tasks.

5.3 Heap Management

The heap memory model used by GATE is based on our earlier work on heap ADT's [15]. Heap ADT's had the restriction of only being able to store one type of data for every mini-heap. This restriction minimised the heap management overhead. The heap memory model eliminates this problem by shifting most of the heap management to the runtime system.

The memory models used by Split-C [16] and UPVM [12] are similar to GATE's heap memory model, as they both use user-level code to access a reserved global-address space. The major differences between Split-C's model and GATE's model are:

- Split-C reserves its partitions for processors (coarse-grained execution units), while GATE reserves its partitions for tasks (medium-grained execution units)
- Split-C uses its memory partitions as a shared resource, whereas GATE ensures private use for its memory partitions by the associated task.

UPVM's model is closer to GATE's model. The major differences between the two models are:

- UPVM requires the use of special memory allocation/deallocation routines.
- With UPVM, every time a memory region is allocated from the global memory space, this region must be reserved on all processors. With GATE, the global address space is partitioned into fragments (called mini-heaps) and each processor owns a collection of these mini-heaps. When a memory region is allocated, it is taken from the processor's collection of mini-heaps and there is

thus no need to convey this request on to all of the other processors.

5.4 Optimising the Migration Transfer

The seed task concept is our solution to optimising the migration transfer procedure. In its most primitive form a seed task is similar to a Remote Procedure Call [17], ie. it is a representation of a function that is about to happen rather than one that has already been activated. Unlike an RPC, a seed task has no specific destination - a load balancing policy determines if and where it is sent. Another difference is that the transfer of a seed task from one processor to another is a delayed action, whereas the transfer of a remote procedure call takes place almost immediately.

A seed task is also similar to Cilk's [18] closure concept. Cilk programs are written using a functional paradigm and a closure can be activated once all of its arguments are available. Cilk's closure concept is only suitable for a functional programming paradigm. We foresee that the seed task concept has the ability to be suitable for multiple programming paradigms.

6. Future Work

6.1 Dynamic Heap Model

The heap model described in this paper is a static task heap model, ie. each task has a single fixed size heap. Its simple structure ensures minimal management overheads. However, it has an obvious disadvantage of wasting the heap space for smaller tasks. To eliminate this disadvantage, a dynamic heap model is under investigation. The new heap model will allow tasks to start with a smaller heap, and the heap size can grow when required. Also, when a host is running out of heap space, it can borrow mini-heaps from other hosts. Thus, making the heap usage more flexible and more efficient.

6.2 Seed Tasks

GATE has provided the mechanism to realise seed tasks. However, to fully utilise the performance benefits that seed tasks have to offer, well-thought-out policies are needed. These policies would determine:

- which tasks would be created as seed tasks
- the delay period between a seed task being created and the same seed task maturing into an active task.

These policies would be based on some dependency relationship between tasks and would allow lazy-activation of tasks, ie. a task would not be activated until it was needed. The Gardens team has already started to work on this topic and the results from this work should be made available later this year.

6.3 File I/O

We intend to enhance GATE to allow file access to be preserved when a task migrates. Gardens assumes the presence of a network file system. At migration time, live file descriptors will be relocated to the task's new location and manipulated so that they can continue to access the file.

7. Conclusion

In this paper we presented a kernel whose design has heavily been influenced by the migration process. It offers several features that provide performance gains and make the migration process a lot simpler.

It provides a medium grained unit of migration that has both performance and ease of use benefits. Performance benefits include a migration unit that has a relatively small state - transfer of state is thus reasonably fast - and low communication overheads between lightweight computational objects (threads and objects). Ease of use benefits includes a migration unit state that is organised as one contiguous memory block - thus only one memory object needs to be transferred.

It provides a strategy for dealing with the problem of communications occurring during the migration process. This strategy, called global synchronisation, offers a simpler and safer solution over existing approaches.

It provides a heap manager that is concerned about migration and uses a distributed memory model that has both performance and ease of use benefits. Performance benefits include the ability to move a task's heap from one processor to another without needing to adjust the pointer values. It provides a facility that is easy to use, as it is does not have to deal with the cache coherence problem.

Finally, it provides seed tasks that are both cheaper and easier to migrate than active tasks, as they have a much smaller state space than active tasks and their migration does not require global synchronisation.

Acknowledgments

Our thanks to both Clemens Szyperski and Paul Roe for their supervision of this work.

The work reported in this paper has been funded in part by both the Cooperative Research Centres Program, through the Department of the Prime Minister and Cabinet of Australia, and the Australian Research Council (via ARC Large Grant 7303). We would also like to acknowledge the QUT Programming Languages and Systems research concentration for providing the infrastructure to implement our system and conduct our experiments.

References

1. P. Roe and C. Szyperski, Gardens: High Performance Objects, Tasking and Migration for Cluster Computing, Technical Report, FIT-TR-97-03, Dept. of Comp. Sci., Queensland University of Technology, Brisbane, May 1997.
2. T. Von Eicken, D. Culler, S. Goldstein and K. Schauser, Active Messages: A mechanism for integrated communication and computation, *Proceedings of the 19th International*

Symposium on Computer Architectures, May 1992, pp. 256-266.

3. N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic and W. K. Su., Myrinet: A Gigabit per second Local Area Network, *IEEE-Micro*, Vol. 15, No.1, February 1995, pp. 29-36.
4. E. Roush, The Freeze Free algorithm for Process Migration, PhD Thesis, University of Illinios, Urbana-Champaign, Illinios, 1995.
5. F. Douglis and J. Osterhout, Process Migration in the Sprite operating system, *Proceedings of the 7th International Conference on Distributed Computing Systems*, 1987, pp. 18-25.
6. D. S. Milojicic et. al., Task migration on the top of the Mach microkernel, *Proceedings of the 3rd USENIX Mach Symposium*, Sante Fe, April 1993, pp. 273-289.
7. E. Jul, H. Levy, N. Hutchinson and A. Black, Fine-Grained Mobility in the Emerald System, *ACM Transactions on Computer Systems*, Vol. 6, No. 1, February 1988, pp. 109-133.
8. J.K. Bennett, Distributed Smalltalk, *Proceedings of the 2nd ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, publ. ACM Press, New York, 1987, pp. 318-330.
9. H. Assenmacher, T. Breitbach, P. Buhler, V. Hubsch and R. Schwartz, The PANDA System Architecture, *Proceedings of the 4th Workshop on Future Trends of Distributed Computing Systems*, Lisbon, Portugal, 1993, pp. 470-476.
10. M. Litzkow and M. Solomon, Supporting checkpointing and process migration outside the UNIX Kernel, *Proceedings of the USENIX Winter Conference*, San Francisco, January 1992, pp. 283-290.
11. J. Casas, D.L. Clark, R. Konuru, S.W. Otto, R. Prouty and J. Walpole, MPVM: A Migration Transparent Version of PVM, *Computing Systems*, Vol. 8, No. 2, Spring 1995, pp. 171-216.
12. R. Konuru, J. Casas, S. Otto, R. Prouty and J. Walpole, A User-Level Process Package for PVM, *Proceedings of the 1994 Scalable High Performance Computing Conference*, Knoxville, Tennessee, May 1994, pp. 48-55.
13. M.L. Powell and B.P. Miller, Process Migration in DEMOS/MP, *Proceedings of the Ninth ACM Symposium on Operating Systems Principles* , 1983, pp. 110-119.
14. J. Shirley, W. Hu and D. Magid, *Guide to Writing DCE Applications 2nd Edition*, Publ. O'Reilly & Associates Inc., 1992.
15. S.Y. Chan and P. Roe, Communicating Complex Data Structures, *Proceedings of the Third Australasian Conference on Parallel and Real-Time Systems (PART'96)*, September 1996, pp. 138-143.
16. D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken and K. Yelick, Parallel Programming in Split-C, *Proceedings of the 1993 Supercomputing Conference*, November 1993, pp. 262-273.
17. A.D. Birrell and B.J. Nelson, Implementing Remote Procedure Calls, *ACM Transactions on Computer Systems*, Vol. 2, No. 1, February 1984, pp. 39-54.
18. R.D. Blumofe and P.A. Lisiecki, Adaptive and Reliable Parallel Computing on Networks of Workstations, *Proceedings of the USENIX 1997 Annual Technical Symposium*, Anaheim, California, January 1997.