Communication of Complex Data Structures

Siu-Yuen Chan

School of Computing Science
Queensland University of Technology
Brisbane, Queensland, 4001
Australia

si.chan@student.qut.edu.au

Paul Roe

School of Computing Science
Queensland University of Technology
Brisbane, Queensland, 4001
Australia

p.roe@qut.edu.au

Abstract

Communicating complex data structures, that is those containing pointers, across machines is a common problem in distributed and parallel computing; particularly with the current move towards object-oriented programming. Some techniques utilise dedicated hardware and system software to efficiently achieve this. This paper presents a study of several software techniques for the communication of complex data structures. A selection of packing/unpacking techniques are tried and compared; in addition a mini-heap ADT, requiring no packing or unpacking, is studied.

1 Introduction

Parallel computing is currently a hot topic in computing. The basic driving force of parallel computing is the desire and prospect for increased performance over conventional sequential computing. Of particular interest to us is network based parallel computing and the idea of creating a virtual supercomputer from networks of idle workstations [3].

Over the past decade the development of parallel computing has shifted from processors with physical shared memory to machines with distributed memories. Shared memory as an interprocessor communication technique is not directly supported by distributed memory machines. Instead, distributed shared memory (DSM), which is the abstraction of shared memory on multicomputer architectures, has been developed for parallel systems of this type. However, generally DSM requires hardware, specialised OS support or at least a homogeneous collection of processors; none of which are available for our virtual supercomputer.

Message-passing is another common interprocessor communication technique and is naturally supported by distributed memory machines. Sun developed XDR to communicate commonly used simple and structured data types in RPC[2]. The Mach distributed operating system tags each item of data in a message with its type [1]. However, neither of these message passing systems directly support the communication of complex data structures containing pointers. This paper investigates high level message-passing, in particular the tradeoffs involved in different techniques for communicating complex data structures containing pointers. This is becoming increasingly necessary with the move towards object-oriented(OO) parallel programming: objects by their very nature utilise pointers. Several other researchers have investigated this tricky problem including Toyn[8] and Newcomer[6]. The efficiency of Toyn's binary transfer relies on the existence of copying garbage collector. Newcomer's algorithm was designed for IDL objects running on distributed systems. Our investigations seek general solutions for the communication of complex data structures for parallel, rather than distributed, computing.

There are two basic issues involved in the communication of complex data structures containing pointers:

1. compaction of data for efficient communication and
2. relocation of data (pointers).

Typically it is much more efficient to communicate a block of contiguous memory between processors than to communicate multiple fragments of memory: for example as separate communications. Basic data types, arrays and records have a contiguous representation in memory and are relocatable (providing they contain no pointers). However, usually complex data types such as linked lists are incrementally constructed and hence are not contiguously allocated in memory. They also contain pointers which typically will not be valid on the destination processor.

The techniques investigated address these issues in different ways. The most straightforward technique is to pack data on communication into a buffer and simultaneously compute relocation information. The buffer and relocation information are communicated. On receipt the data is unpacked and relocation information is used to

reallocate data at different locations in memory. We term this technique packing/unpacking; several variations of this are investigated in the following section.

An alternative technique is to build and maintain data in a compact and relocatable form; the mini-heap ADT, described in Section 3 implements this. The system implement the management of a number of mini-heaps, into which complex data to be communicated is built. It is similar to the collection of the Euclid and Turing [5] programming languages. When a communication request is raised, the whole block of mini-heap data can be directly communicated. This is restricted to communication across homogeneous machines.

2 Packing and unpacking

To generalise packing and unpacking, first we assume all complex data structures containing pointers can be represented by a directed graph. Secondly, we consider only rooted, connected and finite graphs. Under these assumptions, the nodes of a graph represent data records and the edges of a graph denote the pointers that reference those records. To be more general, a graph may contain different types of nodes, i.e. each node of the graph may contain different types of data and have different number of pointers to other nodes of the graph. To achieve this, a tag is given to each node of the graph to distinguish its node type. All the implementations discussed in the following sections are based on these assumptions.

2.1 Thread safety

Thread safety is an important issue when considering different approaches to communicating complex data structures. Different packing/unpacking approaches support different levels of concurrency, and require different locking strategies. A minimum requirement is that no thread writes to a graph while another thread packs the graph. We classify the concurrency of access to the graph to be packed thus:

1. There is no change to the graph, other threads can read or pack the graph.

2. Other threads can read, but not pack, the graph. (This requires an additional mode of locking for graphs: 'being-packed'.)

3. No other threads can access the graph while it is being packed.

2.2 Technique I: node marking

The first approach that we used was to pack and unpack graphs based on the technique described by Griesemer [4]. In this approach, a buffer is used to accumulate the packed graph and a mark is given as an extra field to each node in the graph. The graph is traversed and packed into a buffer for transmission. The purpose of node marks are to mark whether nodes have been reached before in the traversal, and if so at what point in the traversal they were first encountered. All the node marks are initialised to zero. During graph traversal, nodes or references to previously packed nodes are written into the buffer. If it is the first time that a node is reached, its tag and data values are written to the buffer. Also the traversal sequence number is written to the mark of the node in the graph. If the node has been reached before (its mark is non-zero) the negative value of the mark is written to the buffer: denoting a reference to a previous node in the buffer. When a graph is rebuilt, graph nodes are recreated according to nodes' tag values. If a negative value appears after all the data items of a node, we know this node has a reference to a previously created node.

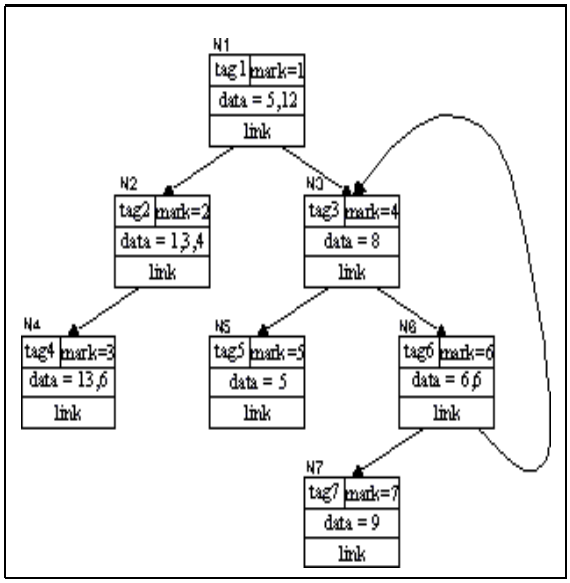

Figure 1: *Example graph data structure.*

For example, consider the above data structure. After the data structure is packed into a buffer, the buffer will contain the following:

tag1, 5, 12	tag2, 1, 3, 4	tag4, 13, 6	tag3, 8
tag5, 5	tag6, 6, 6	tag7, 9, -4	

Since this approach modifies the mark field of the traversing node, no other threads can concurrently pack the graph. However, the graph is still available for read access by other threads.

2.3 Recursion removal: pointer reversal

Conventional graph traversal may be simply programmed using recursion. However, this uses an amount of stack space proportional to the maximum depth of the graph. This could be large

in the case of, for example, a linked list. An alternate approach to using recursion is to use *pointer reversal*. This approach allows iterative traversal of an arbitrary graph, in constant space. During graph traversal, we flip pointer references to point back to previous nodes. These 'back pointers' direct the traversal back to previous nodes when the traversal needs to return to them. The following diagram shows the pointers in the traversal of a simple data structure using pointer reversal:

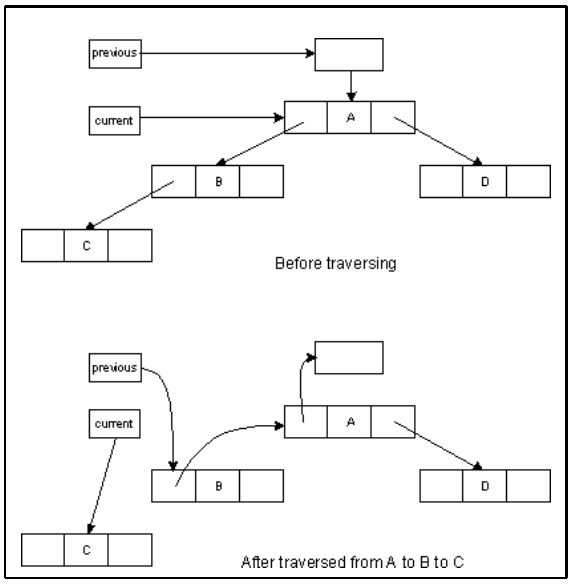

Figure 2: *Pointer reversal traversing.*

Since the links between nodes are changed during the packing, the entire graph structure is changed, hence the graph should be locked and no other threads can have read access to the graph during packing.

2.4 Technique II: index table

Packing and unpacking with the aid of an index table is the most traditional approach to graph packing. It has the lowest efficiency and highest overhead due to the extra index table required. In our approach, the index table serves two purpose. First, it indicates whether a node has been reached before during the traversal. Second, it gives the reference to the pointers relocation when the graph is being unpacked. During the traversal, we pack into the buffer each node's address, tag and data items. Also we write to the index table the current nodes' address and pointers; thus the index table stores node linkage information. This is required as pointer relocation information for the receiver process to rebuild the graph structure. The index table has the form:

current-address	referencing-address

When the graph is rebuilt, first we recreate all the nodes of the graph according to the nodes informations in the buffer. Then we link all the nodes of the graph according to the information in the index table.

Taking the data structure of Figure 1 as an example, after the packing, we have the following byte stream inside the buffer:

N1adr, tag1, 5, 12	N2adr, tag2, 1, 3, 4
N4adr, tag4, 13, 6	N3adr, tag3, 8
N5adr, tag5, 5	N6adr, tag6, 6, 6
N7adr, tag7, 9	

The index table contains all the node reference pairs for relocation of pointers during graph reconstruction.

N1adr	N2adr
N1adr	N3adr
N2adr	N4adr
N3adr	N5adr
N3adr	N6adr
N6adr	N7adr
N6adr	N3adr

The buffer and index table can be combined or sent separately to the destination processor.

The size of data which must be communicated is much larger than for the node marking technique. For an arbitrary graph that has arbitrary edges per node, this difference is hard to estimate. However, if there are N nodes in the graph, and each node of the graph has L edges to other nodes, then the difference between the data size of this approach and the node marking approach will be:

packed data difference + size of index table
= N * address size + 2 * N * L * address size
= N * address size * (2L + 1)

This size directly affects the size of the pre-allocated I/O buffer required and the transfer time required.

The index table lookup time is also a large overhead during the packing and unpacking processes. To optimise this, three different implementations of index tables were tried. The first implementation uses a simple linked-list as an index table. The linked-list table has a small lookup time overhead when the size of the data structure to pack is small. The second implementation uses a m-way tree as index table. This m-way tree is constructed as tries and spends an almost constant time on each table lookup, and is independent of the data structure size. The third implementation uses a hash table as index table [7]. It uses simple mathematical functions to perform the lookup. In our implementation, we use simple double hashing to perform insertion and lookup. Rehashing can be implemented if the size of graphs to be packed is very large. However, the rehashing introduces an extra time overhead to the packing. To avoid this,

it is advantageous to know the size of the graph to be packed a priori so that an appropriate size of hash table can be pre-allocated.

The linked-list implementation of an index table has a size directly proportional to the number of nodes(N) and links(L) per node. If all nodes have same number of links to other nodes, the size of the index table can be calculated by:

$$3 * N * L * \text{address size}$$

By the same assumption, and to guarantee less than 60% population, the the hash table should be larger than 1.5 * N and the total index table size will be:

$$3N * (1 + L) * \text{address size}$$

3 Mini-heap ADTs

Any data structure in memory can be communicated to a compatible machine by simply sending its entire memory area. The receiver must store this stream of data at the same memory address to guarantee that pointers are valid. Usually this is an inappropriate approach because data structures will be distributed over a large memory area and sending this entire area would be very wasteful in terms of time and space.

In our approach we assume that we can maintain an identical address space of the communicating memory block between the sender and receiver process. With this assumption, the receiver process can dereference the pointers correctly. Assuming a SPMD programming model, homogeneous machines and static linking, program code and data segments will be located at the same locations in virtual memory. Thus we may partition virtual memory between processors. By partitioning virtual memory across processors we can guarantee that a receiver can store a block of memory at the same address as the sender, and physical memory is not wasted by partitioning across machines. This is achieved by allocating a large amount of virtual memory at the beginning of program initialisation and partitioning it into several mini-heaps. Thus, we can have the virtual address space of ith mini-heap of the sender process identical to the virtual address space of the ith mini-heap of the receiver process.

To address the problem of data distributed over a large area within the memory, we restrict each mini-heap to represent a single data structure and to store only one type of data. Thus each mini-heap can be further partitioned into a number of equal size storage elements, thereby simplifying storage management.

The mini-heap ADT uses a free list to manage storage elements. When an allocation of memory for this type arises, the mini-heap ADT finds the first free storage element from the free list. Then it allocates this storage element to the variable that raised the request. In order to free elements a garbage collector is required and to maintain compaction this should be a compacting collector. (This was not implemented.) Clearly more sophisticated storage management is possible, for example mini-heaps could be chained together etc. Currently, a explicit dispose element function is implemented for simple management; the freed element will be returned to the free list.

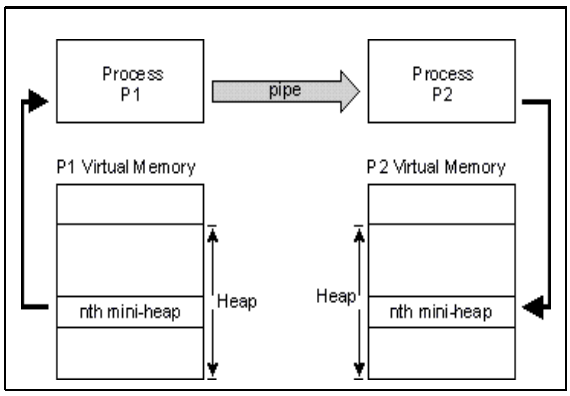

Figure 3: *Communication of a mini-heap.*

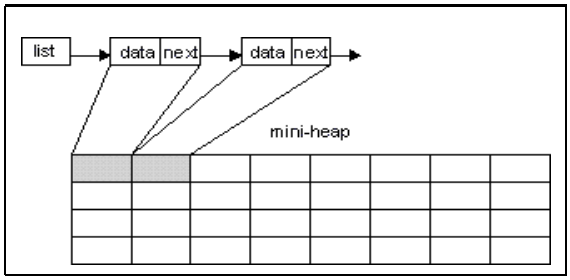

Figure 4: *Build data into mini-heap.*

4 Results and comparison

So far we have introduced five different packing/unpacking techniques and a mini-heap ADT. To carry out the performance tests, we used these techniques to pack and unpack data structures of different behaviour. The mini-heap ADT was implemented but not compared with the packing/unpacking approaches, because its performance is application dependent: see Section 5.

The test programs were written in Oberon, an OO language in the Pascal family, and compiled using a local Oberon compiler (Gardens Point Oberon). The platform was a Sun SPARCstation 4 with 32MB of memory, under Solaris 2.5.

There are some basic time overheads for the five packing/unpacking approaches. The identified basic time overheads are:

1. The reset of node marks in the node marking approach. The node marks must be reset after each packing operation, so that subsequent

packing operations will not be affected. This reset requires another traversal of the graph. So the overhead is approximately half of the packing time.

2. The initialisation of the index table. This overheads can be measured by packing an empty graph. The overhead is negligible on the linked-list and m-way tree table implementation, but is significant on the hash table implementation. A fixed size hash table can eliminate this overhead.

To compare the performance of the five packing/unpacking approaches, we packed and unpacked data structures of different sizes and topologies. The data structures we used were: simple linked-lists, m-way trees, and graphs with various topologies. The chart in Figure 5 shows the timing results of packing a 8-way tree with the five approaches. This set of results is chosen because it is representative of all the results obtained.

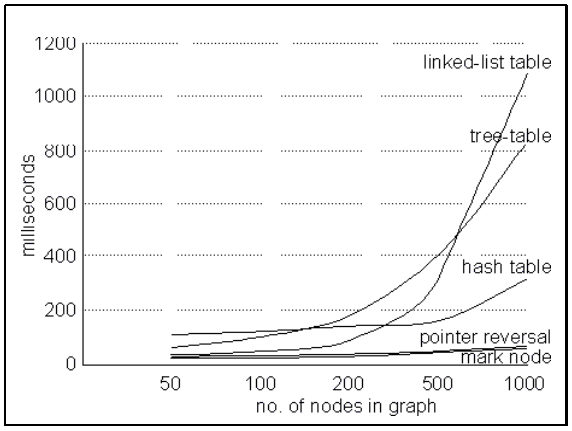

Figure 5: *Time required to traverse a graph with 8 outgoing edges per node.*

4.1 Node marking: recursive vs pointer reversal

Theoretically, by saving on procedure calls, the pointer reversal approach should have less time overhead than the recursive node marking approach. However from the chart shown in Figure 5 we can see that the time overhead of the node marking and pointer reversal approaches are almost the same. This is because the pointer reversal approach needs to perform more pointer manipulations than the recursive approach, which introduces time overheads that compensate for procedure calling. Both of these approaches have a linear and gently increasing time overhead with respect to the increasing graph size; since there is no table lookup operation, which may cause the polynomial time overhead behaviour as with the index table approach.

4.2 Thread safe approaches

The linked-list index table implementation has a low time overhead when the size of the packing graph is small, and is comparable to the node marking and pointer reversal techniques. However, this time overhead increases rapidly as the graph size becomes large: $O(n^2)$. The hash index table implementation has the largest initial time overhead among the three thread safe approaches. It increases gently as the graph size increases and is good to use if the graph size is large. The characteristic of the tree index table implementation is *O(n log n)* which falls between that of the linked-list and hash index table implementation.

As mentioned before, the node marking and pointer reversal approaches are thread unsafe because they modify the data structure during the packing operation. These two approaches, however, have the lowest time overhead among the packing approaches. The only thread safe approach that has time overhead close to these two for large graph sizes is the hash index table implementation.

A characteristic of the hash index table implementation is its time overhead is affected by the population of the hash table. If the hash table is fully populated, the lookup overhead may be increased due to the fact that secondary hash is required when there is collision on primary hash, and the lookup overhead may become $O(n^2)$. So it is necessary to carefully adjust the size of the hash table to minimise this extra time overhead.

5 Conclusions

The performance tests show that for packing and unpacking using a node marking technique is far more efficient than using an index table. However it has the shortcoming of not being thread safe because it modifies the node marks during the graph traversal. It is also necessary to reset the node marks to zero before the next traversal of the graph.

Although the index table approach has the lowest efficiency and highest overhead, it does have the advantage of being thread safe because it does not modify any field of any node in the graph. It is safe in the sense that it does not cause inconsistency to other processes reading the data structure. Also the performance tests show that most of the time overhead is index table lookups to determine whether a node has been reached before. This overhead is directly related to the implementation of the index table.

The pointer reversal traversing approach saves stack space allocated during the traversal of a graph; however, it modifies pointers as it traverses through the graph, therefore it is also not thread

safe. It can be used together with the node marking approach. We did not use it with the index table approach because we do not want to give up the thread safe behaviour of the index table approach.

The performance of the mini-heap approach is algorithm dependent. The frequency of an algorithms allocation and deallocation will determine how fragmented the mini-heaps will become and hence how efficient this approach will be. This approach addresses the problem in a simple manner and can sometimes provide an efficient transfer of large complex data structure. The restriction is it can be used only on homogeneous platforms.

Each of the approaches studied in this paper has its advantages and disadvantages. An important criteria is whether thread safety is required; if not an efficient packing algorithm can use pointer reversal and node marking. Where thread safety is required the index table represents the greatest source of inefficiency; further work is required in order to optimise this. Mini-heap ADTs can in some cases provide an efficient implementation, but this is rather application dependent. In particular it depends on the nature of the data structures used in the application. If graph nodes are infrequently allocated and deallocated with respect to the frequency of graph communication it may be a useful approach.

For the Gardens project it is hoped to use some of these techniques for generic object communication.

Acknowledgements

We would like to thank the anonymous referees for their comments on an earlier draft of this paper. This study has been supported by the Gardens research project at QUT.

References

[1] J Boykin, D Kirschen, A Langerman, and S Loverso. *Programming Under Mach*, chapter 3, pages 63–97. Addison-Wesley, 1993.

[2] J R Corbin. Sun technical reference library. In *The Art of Distributed Applications: Programming Techniques for Remote Procedure Calls*, chapter A2.3, pages 262–274. Springer-Verlag, 1990.

[3] J Diederich, J Gough, G Mohay, and C Szyperski. The Gardens Project—an introduction. In *Australasian Computer Architecture Workshop*, Adelaide, January 1995.

[4] R Griesemer. On the linearization of graphs and writing symbol files. Technical report, ETH, Zurich, March 1991.

[5] R C Holt, P A Matthews, J A Rosselet, and J R Cordy. *The Turing Language: Design and Definition*. Prentice Hall, 1987.

[6] J N Newcomer. Efficient Binary I/O of IDL Objects. *ACM SIGPLAN Notices*, 22(11):35–43, November 1987.

[7] R Sedgewick. *Algorithms in Modula-3*, chapter 16, pages 231–244. Addison-Wesley, 1993.

[8] I Toyn and A Dix. Efficient binary transfer of pointer structures. *Software Practice and Experience*, 24(1), November 1994.