Massively Parallel Garbage Collection

Jeremy H. Brown

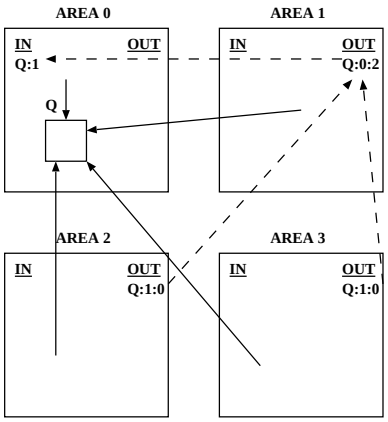
The Problem: We are investigating the problem of garbage collecting a gigantic heap on a massively parallel computer architecture. The target architecture is a distributed memory multiprocessor with a high-speed communication network. Each node features a processing element, a relatively small local memory (1-4 megawords,) and a disk to enable a large virtual memory; there are no data caches. The programming model features both objects which reside in exactly one node's memory and "faceted" objects which may have regions of memory ("facets") in multiple nodes.

Garbage-collection for this architecture must not send too many messages on the inter-processor network, nor generate excessive paging to and from disk. It must generally rely on incremental garbage-collection that does not require system-wide synchronization. It must allow objects to be moved both within their home node's address space and to other nodes.

Motivation: One of the greatest difficulties faced by programmers has always been that of memory management; garbage-collected languages such as Lisp and Java are generally understood to enable significaly shorter program development times than those required for programs written in Fortran, C and C++. One of the major stumbling blocks faced in large-scale parallel computing has always been the difficulty of programming parallel computers. Based on these observations, we view implementing efficient garbage collection as a critical part of developing a generally useful, massively parallel computer system.

Previous Work: [1] is an excellent reference work covering garbage collection techniques ranging from simple reference counting to elaborate distributed algorithms. In traditional reference counting, each object maintains a count of the number of pointers to it; an object with a reference count of zero may be garbage collected. In a distributed setting in which reference counting is taking place only on pointers between objects in different "areas", this means each area must communicate directly with an object's home area to adjust the reference count; this leads to synchronization issues on every pointer copy.

Indirect reference counting [3] (IRC) is designed to avoid the need to update the count in the object's home-area when a pointer is copied between two other areas: when a pointer P is copied from area A to area B, area A increments a reference count associated with P, and area B records the fact that the pointer came from area A – it doesn't matter which area P points into. Thus, the total reference count of the object pointed to by P is represented by a tree whose structure mimics the inter-area diffusion pattern of P: the root of the tree is the area containing P's target; child nodes point to their parents. The following figure illustrates the relationship between pointers and the IRC diffusion-tree.



Indirect Reference Counting: *pointer copied from Area 0 to Area 1, then from Area 1 to Areas 2 and 3.*

When all copies of P are finally eliminated from area B and B's reference count for P reaches zero, B sends a message to A decrementing A's reference count for P. Thus, exactly two messages are needed for

each inter-area pointer copy: one for the copy itself (which also serves as the reference increment message), and one for the decrement when the copy and all sub-copies have been destroyed. Algorithmic synchronization is essentially automatic.

Pointer swizzling ([4], [2]) is a technique originally invented for mapping large heaps into small virtual address spaces. Objects are stored with opaque (indirect) object identifiers (OIDs) which are translated into virtual memory addresses (swizzled) when they are brought into main memory. For our purposes, the main advantage of swizzling is that objects can be arbitrarily relocated when there are no direct pointers to them.

Approach: We have designed a garbage collection system which we are presently implementing in simulation. Our scheme extends indirect reference counting to handle faceted objects. Each node provides a number of independent areas: a small number of "active" areas which are in physical memory, and a potentially large number of "inactive" areas which reside on disk. Objects which go unused are eventually collected together into a new area which is then made inactive.

Pointers between active areas are direct; when an area deactivates, any inter-area pointers it contains are swizzled to refer to canonical object identifiers which do not change as objects are relocated due to compaction or migration.

To study our GC scheme, we are simulating a highly abstract parallel stack machine. By avoiding detail in the simulator, we are able to simulate a machine with hundreds of processors, and to execute much more complex programs than we could on a more detailed simulator. Our goal is to generate realistic event counts for several interesting programs; event counts will include memory references, pages swapped in from or out to disk, number and lengths of messages sent on the interprocessor network, etc. and will be assigned either to user code execution or garbage collection overhead. We will use these counts, in conjunction with simple cost models, to evaluate the costs and benefits of our garbage collection scheme.

Future Work: We intend to extend our simulations to evaluate the benefits of adding hardware mechanisms to assist with GC. For instance, we will evaluate the utility of reference-count caches that collect increment and decrement information for frequently-referenced objects without requiring access to larger tables stored in primary memory.

At the moment our GC scheme does not address the problem of inter-node garbage cycles. Because massively parallel systems have the potential to generate extremely long serial data structures, incremental mark/sweep algorithms may not complete in a reasonable amount of time; we are therefore very interested in developing conservative, parallel algorithms which complete more quickly.

Research Support: Support for this research was provided in part by the Defense Advanced Research Projects Agency under Rome Labs Contract Number F30602-98-1-0172.

References:

[1] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Dynamic Memory Management*. John Wiley & Sons, 1996.

[2] J. Eliot B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering (TSE)*, 18(8):657–73, August 1992.

[3] Jose M. Piquer. Indirect reference counting: A distributed garbage collection algorithm. In *PARLE '91 Parallel Architectures and Languages Europe*, pages 150–165, June 1991.

[4] Paul R. Wilson and Sheeta V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *International Conference on Object Orientation in Operating Systems*, pages 364–77. IEEE Computer Society, IEEE Press, Sept 1992.