

# Hidden cost of memory management in asynchronous communication: A Distributed Hash Table Example

J.M. Malard and R.D. Stewart\*

March 15, 2002

## Abstract

Dictionaries in general, and hash tables in particular, are an essential component of many computational science and data-mining applications. It is shown that in the context of a distributed asynchronous hash table on top of network of IBM SMPs, the main burden performance drag related to memory management is waiting for buffers to become de-allocatable. A simple queue mechanism is found to be effective.

## 1 Introduction

Dictionaries in general, and hash tables in particular, are an essential component of many computational science and data-mining applications. Several major sequential scripting and programming languages such as Python and Java, incorporate built-in dictionaries. For any distributed hash table API a choice must be made as to whether or not internal storage and buffer space will be provided by the user. One approach, typical of fortran 77 programming, is to pass all the necessary workspace from the caller through the library API. The apparent advantages of this approach are many. First, the user knows exactly how much space will be used; although an API could set an upper limit on the amount of dynamically allocated memory. Second, the user may use this workspace for other purposes in between calls to the hash table library; yet anyone involved in porting Fortran H code to an MPP is aware of the related portability issues. Third, it takes time to allocate storage dynamically, yet it may not be possible to know in advance how much storage to allocate on each

---

\*Pacific Northwest National Laboratory, Battelle Boulevard, P.O. Box 999, Richland, WA 99352. The Pacific Northwest National Laboratory is operated by Battelle Memorial Institute for the U.S. Department of Energy under Contract DE-AC06-76RLO 1830.

process when the distribution of the hashed keys is unknown. A fourth argument might be that memory management along with pointer arithmetic are the main source of software bugs. Truly, all these arguments are valid; a hash table interface that addresses many of them and yet allocates its own buffer space is presented in [4]. The purpose of this paper is to expose the impact of memory management on this library in the context of synthetic benchmarks. It turns out that the main runtime drag comes not from memory allocation but from waiting time associated with memory de-allocation.

Our hash table library is implemented on top of IBM's LAPI active messages. For a detailed discussion of active messages see for example [3, 5]. The main reason for using active messages is the high synchronization cost imposed in this context by traditional communication APIs such as MPI [2], SHMEM or the Advanced Remote Memory Copy Interface (ARMCI)[6]. To see this, suppose, that an origin process **O** requests up to  $N$  values associated with a key **K** from the target process **T**. The number of values returned cannot be determined before the target process is queried. Moreover, for the request to be meaningful the matching values on the target process **T** should be returned atomically to the origin process **O**, lest they are deleted by some other concurrent request. Computations that can potentially use a distributed hash table include sparse matrix gather and scatter, histogram construction in discretized grid numerical computations, data clustering with millions of data records and features, etc.

An active message is a form of remote procedure call that is initiated by an origin process, and that may or may not involve synchronization with the target process. Several implementations of active messages have been developed but few appear to be actively supported at the level of GAMMA [1] and IBM's Low-Level Communication Application Programming Interface (LAPI), see for example [7]. The APIs of these implementations vary widely; typically the origin process of an active message supplies the address of a message handler to be invoked at the target process. For the present work, we used active messages from the IBM LAPI library because it runs on some of the largest and most heavily used computers at PNNL and because of the current lack of alternative active message layers that have acceptable performance and widespread distribution.

## 2 Hash Table Data Structure

Active messages in LAPI are initiated at the origin process by a call to `LAPLAmSend`. The arguments to this procedure include: a message header, a message body and the address on the target process of a header handler function. The message header contains all the information needed by the communication system to deliver the message at the target process. It can also include a limited amount of user information. The header handler returns two things to the LAPI (runtime) system: a pointer to some space where to store the incoming message and the address of a completion handler to process the message once it has been copied to the latter space. The header supplied by the LAPI runtime system to the header handler written by the user is only valid during the execution of the user

header handler. The latter subroutine must therefore make a safe copy of this header for use by the matching completion handler.

The LAPI runtime system involves three threads: a user thread, a notification handler thread and a completion handler thread. Header handlers may be called from any of the two first threads. Completion handlers are exclusively called from the third thread. The design of LAPI makes it possible to relegate all memory management to the user code.

Our distributed hash table library (CSE\_HASH) is implemented in tree nested levels: distributed, threaded and serial. The serial level implements a simple dynamic hash table with open addressing and fixed bucket size. The threaded level simply wraps mutex locks around every operation at the serial level. Mutexes are needed to prevent local accesses to a hash table from interfering with accesses that originated from a different process through a LAPI call.

The implementations of insertion and deletion are described next to show where explicit memory management occurs. Let us first look at insertions. When values  $\mathbf{Vs}$  for a given key  $\mathbf{K}$  are to be inserted by process  $\mathbf{O}$  in the hash table segment at process  $\mathbf{T}$ , the necessary header data, including the address of  $\mathbf{Vs}$ , is saved in a message request at  $\mathbf{O}$ , and an active message is initiated. Upon reception of this message at  $\mathbf{P}$ , the header handler at  $\mathbf{P}$  allocates space for a copy of the header and for the incoming values. The matching completion handler will lock the segment of the hash table at process  $\mathbf{T}$ , it will attempt to insert the said key-value pairs and then it will release its lock. Locking the whole hash table segment each time it is accessed can be justified on the ground that handlers can only run within the completion handler thread, hence one after another. It may be that not all the key-value pairs could be inserted, say because the local hash table has reached mid-way a hard upper limit set by the user. It is therefore important for the origin process  $\mathbf{O}$  to know how many values were successfully inserted at the target process  $\mathbf{T}$ . For this purpose, the completion handler issues a put operation that notifies the origin process  $\mathbf{O}$  of the number of successful insertions. This put operation also increment a remote LAPI counter at the origin process. The completion handler at  $\mathbf{T}$  cannot return before the local data for this last put operation can be safely discarded.

Deletions at the target process  $\mathbf{T}$  issued from the origin process  $\mathbf{O}$  works about the same way as an insertion except for the fact that the key and the associated values are transferred from the target process  $\mathbf{T}$  to the origin process  $\mathbf{O}$ . This means is that space for these values is allocated in the completion handler. It also means that an additional put operation is needed to transfer values to the origin process. This second put can take considerably longer to complete. No other accesses to the hash table at  $\mathbf{T}$  can reuse or release space used by this put operation before an appropriate counter is incremented by the LAPI runtime system.

Three questions arise naturally. What time is spent allocating buffer space on average? What is the relative cost of rehashing a table when it overfills? How much time is spent waiting to free storage?

| $K$ | insertion  |            | deletion   |            |
|-----|------------|------------|------------|------------|
|     | initiation | completion | initiation | completion |
| 0   | 6.58822    | 83.7126    | 7.07645    | 342.28     |
| 1   | 5.91850    | 80.9096    | 128.484    | 424.517    |
| 128 | 6.56755    | 90.7731    | 7.93997    | 103.691    |

Table 1: Breakdown of the average time in micro-seconds for inserting or retrieving a single key-value pair.

### 3 Benchmarking

Runtimes were measured in dedicated mode on an IBM SP at Pacific Northwest National Laboratory with 4-way Power III nodes with 375 MHz CPUs. All the runtimes reported here are for 4 such nodes, totaling 16 processors. Bandwidth estimates are not presented here, for large enough values of  $L$  and in the absence of network congestion, they are similar to that of the LAPI\_Put operation that transfer the values between processors. We study the average performance via the insertion and deletion of  $M = 1,000,000$  keys uniformly distributed in the range from 0 to 7,000,000. The same sequence of keys is to time insertions and deletions. The sequences of key insertions and deletions are block distributed among the  $p$  processes. The mapping between keys and processes is cyclic, in that all values associated with key  $k$  are stored in the hash table segment of process  $P_{k \bmod p}$ . In details, process  $P_i$  inserts  $L$  values with key  $k_j$  with  $k_{M/p*i} \leq j < k_{M/p*(i+1)}$  onto process  $P_{k_j \bmod p}$ . Those  $L$  values are otherwise independent, e.g. it is possible for one process to retrieve half of them, while another process retrieves the remaining half. Accesses to the hash table are blocked in the sense that the origin process waits for completion of all pending requests only after initiating 100 hash table accesses. The impact of this blocking factor is outside the scope of this paper, see [4] for details.

On average a deletion takes significantly more time to complete than an insertion does. The reason is that in the former case the completion handler thread is tied up waiting for the final put operation to finish locally. This is illustrated in Table 1. There, a circular queue of size  $K$  is allocated along with each local hash table. When  $K > 0$ , the completion handler for deletions returns immediately after pushing onto this queue pointers to all the transient information needed for completion of the two final put operations. When the queue is full, the completion handler simply waits on the LAPI counter associated with the put operations at the front of the queue. The front of the queue is then moved by one, storage is released and the completion handler reuses the queue slot that has become last. When  $K$  is 0, the completion handler for deletions waits on its own two LAPI puts before returning. That is the queue is not used. In all cases, a single 64-bit value ( $L = 1$ ) was inserted or deleted with each key. The clock resolution is about 3 micro-seconds.

In the present context, the time spent calling the memory allocation routine

malloc on behalf of the completion handler is small compared to the time spent waiting for arguments to LAPI puts to become reusable or releasable. In the case of the deletions, such waiting ( a call to `LAPI_Waitcntr`) can be up to 40 times longer than the time spent in malloc. In the present context, runtime is not a strong reason for eliminating calls to malloc.

Experiments to related performance with a user defined upper limit on the table density did not produce any significant difference in the maximum completion time for values between 0.1 and 0.9. Significance is measured with respect to the clock resolution of 3 micro-seconds. There was though an impact in the sense that the time spent by each process converged to the maximum as the threshold approached 1.

## 4 Conclusions

It has been seen that in the context of a distributed hash table the major runtime drag due to memory management befalls on memory de-allocation not on memory allocation. Precisely, most of the time at both the origin and target processes can be related to waiting for releasing space already in use. It was shown that a simple circular queue can alleviate this problem when the active message framework makes it reasonable to lock entire local segments of the hash table. This locking strategy may not be pertinent in the context of completion handlers that run on multiple threads, that communicate with third-party processes or in the context of non-atomic values. In all such cases, the addition to the runtime system of a limited garbage collection capacity may be necessary instead of the simple queue presented here.

## References

- [1] G. Chiola and G. Ciaccio. Architectural issues and preliminary benchmarking of a lowcost network of workstations based on active messages. In *14th ITG/GI Conference on Architecture of Computer Systems*, pages 13–22, September 1997.
- [2] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, and M. Snir. MPI-2: Extending the message-passing interface. Technical Report MCS-P568-0296, Argonne National Laboratory, July 1996.
- [3] S.S. Lumetta, A.M. Mainwaring, and Culler D.E. Multi-protocol active messages on a cluster of smp's. In *Supercomputing 1997*, November 1997.
- [4] J.M. Malard and R.D. Stewart. A distributed hash table library based on active messages. In *submitted to SPAA02*, March 2002.
- [5] R.P. Martin. HPAM: An active message layer for a network of HP workstations. Technical Report CSD-96-891, University of California at Berkeley, 1996.

- [6] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. *Lecture Notes in Computer Science*, 1586, 1999.
- [7] G. Shah, J. Nieplocha, C. Mirza, R. Harrison, R.K. Govindaraju, K. Gildea, P. DiNicola, and C. Bender. Performance and experience with LAPI: a new high-performance communication library for the IBM rs/6000 sp. In *Proceedings of the International Parallel Processing Symposium*, pages 260–66, 1998.