Distributed Dynamic Hash Tables Using IBM LAPI

J.M. Malard[1] and R.D. Stewart[2]

Pacific Northwest National Laboratory[3]
Battelle Boulevard, P.O. Box 999
Richland, WA 99352

Abstract

An asynchronous communication library for accessing and managing dynamic hash tables over a network of Symmetric Multiprocessors (SMP) is presented. A blocking factor is shown experimentally to reduce the variance of the wall clock time. It is also shown that remote accesses to a distributed hash table can be as effective and scalable as the one-sided operations of the low-level communication middleware on an IBM SP.

1. Introduction

The hash table is essential to many computational applications and the related literature is huge[27,12]. Hashing may occur implicitly in some applications such as binning in finite element computations with moving adaptive numerical grids[16,17,23,26] or in image analysis during iso-surface construction. Hashing also arises in graph induction and state-space generation[30,6] and in data mining[11]. Traditional numerical computations such as multipole methods and N-Body problems also make use of hashing[28].

This paper is about a software library (called CSE_Hash) for accessing and managing dynamic hash tables over a network of Symmetric Multiprocessors (SMP)[13]. These hash tables are dynamic because they can grow to accommodate additional entries, but also because they can be instantiated remotely on a target process from an origin process. Implementations of active messages that are both documented and actively supported have recently become available[3,5,14,24]. For the present work, we used the Pthread library[18] in addition to the active messages from the IBM LAPI library because both libraries run on some of the large and currently most heavily used computers at PNNL.

The CSE_Hash library has been designed to be a key tool for accessing and managing dynamically allocated memory asynchronously across a network of SMPs. Our primary goal is for this library to serve as a building block for data intensive applications envisioned at PNNL. As such, it is important that the performance of insertions, look-ups and deletions be as close as possible to that of the low-level communication operations with which application scientists would program their hash tables. One can easily appreciate by the size of the current literature on hashing that parallel implementations of hashing do exist. For instance, the Multipol library of distributed data structures from Katherine Yelick et al.[29] contains a general non-blocking distributed hash table. Threads are central to the design of the Multipol system and it was written in terms of active messages that were themselves implemented using message-passing libraries such as NX on the Intel Paragon and MPL on the IBM SP1 and SP2. Multipol applications are written in the Shared Memory Programming Paradigm where all data is uniformly available to all processes. Another related effort, the Hierarchical Distributed Dynamic Array (HDDA) system by Manish Parashar et al.[21,22] was developed in the context of Structured Adaptive Mesh Refinement Applications. Each element of a HDDA can itself be an HDDA. This system maps a hierarchy of lattice points onto an extensible hash table where the mapping between lattice points and

---

[1]Computational Science and Mathematics – email: jm.malard@pnl.gov
[2]Radiation and Biophysics Research
[3]The Pacific Northwest National Laboratory is operated by Battelle Memorial Institute for the U.S. Department of Energy under Contract DE-AC06-76RLO 1830.

0-7695-1524-X/02 $17.00 (c) 2002 IEEE

hash keys is defined by space-filling curves. The mapping between spatial coordinates and hash keys is made efficient by the assumption that the underlying lattice is regular. Hash table entries point to buckets that are joined and split to accommodate insertions and range queries. This adds one level of complexity relative to the hash table studied here where each value is stored in the hash table along with its key. When the hash table is tightly bound to an application or to a programming model it may become difficult understand the performance aspects of dynamic hashing in a parallel environment and to isolate the important algorithmic parameters. The latter is our secondary goal in this work.

The API presented here is less general than say Linda[4] or Codd's relational algebra[7]. Linda has a single tuple-space from where applications can enter and retrieve values. Those values can have any size and may be executable. Values in tuple-space may be retrieved by any process using a pattern-matching mechanism. In comparison, the CSE_Hash library supports one separate tuple-space for each distributed hash table that has been initialized. All values within a hash table are constrained to be sequences of non-executable atomic values of the same type and length. The general Linda framework introduces many software parameters that may affect the runtime performance of remote memory access. The ability to maintain several independent spaces in a multi-threaded environment is an important consideration as well for the complex message-passing applications that were being considered when the CSE_Hash API was defined. The relational algebra includes multiple concurrent spaces in the forms of typed relations. On the other hand, the usual relational operators or joins are not well suited to implement irregular or random access patterns.

On another level, the hash table API described in this paper differs from the Message Passing Interface MPI[8], SHMEM[25], the Advanced Remote Memory Copy Interface (ARMCI)[19] or IBM's Low Level API[24] in that it replaces the concept of a physical address by that of a user specified index. It also differs from array based toolkits and language extensions such as Global Arrays[20] and HPF[10] because those indices need not be contiguous and they are bounded by the admissible range of keys rather than by the physical memory available.

2. Distributed Hash Tables

2.1. An Example: One-sided matrix scatter

The CSE_Hash library assumes a Multiple Instruction Multiple Data Model where independent processes cooperate to compute some operation. For instance, the rows of a sparse matrix *A*, initially stored on one of *p* processes, say P_0, might have to be scattered onto all *p* processes. The rows might be dealt to processes in a round-robin manner (a row cyclic distribution) with the first row sent to process P_1, the second row to P_2, etc. Many ways exist to store a sparse matrix in the memory of a single process. A very simple technique is to insert pairs formed of a column index and the corresponding non-zero value into a hash table using the corresponding row index for the key. Suppose that the non-zero values are stored in a file, one per line, as triplets of corresponding row index, column index and non-zero value. This last storage scheme is known as coordinate storage. It uses about 50% more storage than most list-based schemes but is relatively easy to use. Figure 1 illustrates, using the language C, how process P_0 might read the matrix A into a local hash table denoted *hash_x*. The iteration is over all *nnz* non-zero values of *A*. This number has been read previously from the "input file" *ifp*. The first statement, a procedure call, reads in one non-zero value along with its row and column indices. The next three statements pack the column index and floating point value into a 2-integer array called *batch*. The last statement inserts this array into the local hash table. The denotation *hash_x* is also known as a handle. The statement that creates this hash table would specify that all its values are sequences of two integers. The last argument of the function hash_insert_x is an input/output argument. Upon entry, its value is the number of pairs to be inserted into the table. That is, a single call to hash_insert_x could insert many pairs of integers, as long as they are stored contiguously. It is possible that an upper limit has been set on the maximum size of the hash table and that this limit prevents some pairs in *batch* from being inserted. Upon exit from hash_insert_x, the value of *n* is the number of pairs that were really inserted. This number *n* is important, later it will be referred to as the *request size*. One last point, two threads on process P_0 might try to access the hash table hash_x concurrently. The naive approach to ensuring that accesses are atomic is to wrap a lock around the hash table and that is what is done here. The handle hash_x contains one such lock, which is grabbed by hash_insert_x before entering the hash table and released before exit.

```
for ( i=0; i<nnz; i++ ){
      /*Read one non-zero entry of matrix A.                    */
      fscanf(ifp, "%d %d %f", &row, &col, &val);

      /* Store the column index and non-zero floating point  */
      /* value into and array of integers called batch.      */
      one.f = val;
      batch[0] = col;
      batch[1] = one.v;

      /* Insert this two-element array into the local hash   */
      /* table hash_x                                        */
      n = 1;
      hash_insert_x (hash_x, row, batch, &n);
}
```

Figure 1 How process P_0 might read a sparse matrix in coordinate storage into a local hash table.

The next step for process P_0 would be to remove for its local hash table *hash_x*, those rows that belong to other processes and to insert them into local hash tables of the appropriate target processes. Figure 2 shows a C code fragment for doing just this. Process P_0 initiates some fixed number L of remote insertions and then waits for them to complete. This *blocking factor L* is important and will reappear later. It is being introduced for two reasons. First, space is limited. The buffer space where the inserted values are held, array *batch*, cannot be reused before the transaction is complete, that is before the inserted values have reached the target process. If m denotes the maximum number of non-zero in any row of A then the array *batch* must be able to store at least $2*L*m$ integers. Second, some communication middleware impose a limit on how many communication requests may exist in the system at any one time. Initiating, without pause, a million communications all targeting the same process might stall such middleware.

```
nsent = 0;
for ( i=1; i<=nrows; i++ ) {
      /* Initiation phase starts */
      target = i%p;
      if ( target!=0 ){
            n = m;
            hash_delete_x (hash_a->hash_x,
                  i, &batch[2*nsent*m], &n);

            HASH_Insert (hash_a,
                  target, i, n,
                  &batch[2*nsent*m],
                  &reqs[nsent]);
            nsent++;
      }
      /* Completion phase starts */
      if ( nsent == L || i+nsent >= nrows ){
            HASH_Wait( nsent, reqs);
            nsent = 0;
      }
}
/* Barrier phase starts */
HASH_Flush( hash_a );
```

Figure 2 How Process P_0 might scatter the matrix A using a cyclic row distribution.

The functions for searching and deleting values from the local hash table, hash_find_x and hash_delete_x, have nearly the same calling sequence as hash_insert_x. The only difference is that the array of values is an output argument. Those

two query functions differ from one another in that the values returned by hash_delete_x are no longer in the local hash table when it returns.

The local hash table *hash_x* does not contain all the necessary information to constitute a truly distributed hash table. Instead, *hash_x* is a member of a more complex structure *hash_a.* This relationship is captured in C by writing *hash_x* = *hash_a*->hash_x. The handle, *hash_a*, was returned by the call that created the distributed hash table. Values can be inserted, looked-up and deleted from the distributed hash table by all processes holding a handle that match hash_a. The functions for the latter distributed operations, respectively HASH_Insert, HASH_Find and HASH_Delete, are *non-blocking*; the communication operation they initiate need not be complete before they return. The same calling sequence is used for all the initiation functions HASH_Insert, HASH_Find and HASH_Delete, namely

```
ierr = HASH_Insert ( hash_a, target, key, n, arrayofvalues, &request ).
```

Here, *target* is an integer between *0* and *p-1* that identifies on which process the *n* values stored in *arrayofvalues* will be inserted. We could eliminate this argument if for instance we knew that a cyclic distribution of the keys is always optimal, that is all values for key *k* are inserted on process *k%p* (*k* modulo *p*). The third argument is the value of the key. The fourth argument, the request size *n*, has already been discussed. Note that values assigned to keys are sequences of values of basic data types, e.g. int, long, long long, etc. Those (sequences of) values are stored contiguously in the one-dimensional array *arrayofvalues*, which is an output argument in the case of HASH_Find and HASH_Delete. The last argument is a pointer to a communication request. When this pointer is non-null, the insertion is complete when a matching call to HASH_Wait returns. At that time, the number of key-value pairs processed successfully can be read using the request handle. In the case of HASH_Insert and HASH_Delete, all values transferred from the target process are then available to the origin process in *arrayofvalues*. The last function call in Figure 2 is a call to a barrier. This call is collective, contrary to HASH_Wait. If HASH_Wait is called, it must be called by all processes sharing the handle *hash_a*. It returns when all remote accesses to the distributed hash table are complete.

One last point about Figure 2 is in order before going into the details of how remote hash table accesses are implemented. How should the wall-clock time of this loop be defined? Suppose that the bulk of the work takes place between the repeated calls to HASH_Insert and the call to HASH_Wait. The latter would likely return immediately and all the work would be done when HASH_Flush is called. In that case, the relevant wall-clock time is be the sum of the times spent in HASH_Insert and hash_delete_x. Let us call this time the *initiation time*. If on the other hand, the bulk of the work were done inside the loop but after HASH_Wait returns, the relevant wall-clock time would be the sum of the initiation time and of the time spent in HASH_Wait. Let us call this time the *completion time*. Finally, when the bulk of the work is done after the barrier HASH_Flush is called, then the relevant wall-clock time is the sum of the completion time and of the time spent in HASH_Flush. Let us call this time the *barrier time*.

2.2 Execution model

As for the details: when a value is inserted remotely by an *origin* process in the local table of a *target* process, the origin process initiates a so-called active message whose purpose is to invoke a remote procedure or handler on the target process using the data supplied to the active message. An active message then consists of three main components: the identity of the target process, the address of the handler on the target process and the arguments to this handler. In principle, active messages can be implemented without threads using simple point-to-point message passing and some appropriate data structures. In some implementations, a new thread is spun on the target process each time an active message arrives. In other implementations, a service thread handles all in-coming active messages. The LAPI implementation is of the latter type. The information associated with an active message is divided into packets. The first such packet, or message header, contains all the information needed by the communication system to deliver the message at the target process; it may also include a limited amount of user information. The handler routine is split into a header handler and a completion handler that are both written by the LAPI user. The header handler returns two things to the LAPI (runtime) system: a pointer to some space where to store the incoming packets and the address of a completion handler to process the message once it has been copied to the latter space. The LAPI infrastructure 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 LAPI active message framework is lightweight; no new thread is spun when an active message is received at the target, and there is no garbage collection of buffers allocated by either of the user handlers. Returning to the example of a remote insertion, the header handler has allocated enough buffer storage for the user data and called the completion

handler. The completion handler inserts the new values into the local hash table on the target process and then sends the status of the transaction back to the origin process. This status consists of a completion flag and the number of entries successfully entered in the table. This information is sufficient for the origin process to determine if and which values were not inserted, for instance because for instance the local table size reached a preset limit. It is natural to implement this last exchange using a one-sided put operation. The last task of the completion handler is to release the buffer space allocated by the header handler. This cannot be done while those buffers are being used to return data to the origin process. Experiments with an earlier distributed hash table showed that up to 40% of the time spent in the completion handler was spent waiting to deallocate buffers. The simple cure is to keep a *message queue* where the completion handler can leave the state of the remote insert. When a completion handler is called, its first step is then to wait for completion of the first request on the queue and to deallocate the corresponding buffer space. The maximum number of states that can be saved in this message queue is important and it is denoted by the letter q.

2.3 A note on the impact of threads

In summary, the CSE_HASH library is implemented in three 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 Pthread mutexes around every operation at the serial level. The distributed level relies on an active-message layer to initiate the appropriate thread-level operation on the target process and on one-sided communications to return the status of the operation to the origin process. What then is the cost of thread management? That cost is higher than that associated with mutexes only; for instance the invocation of a completion handler can provoke an instruction cache miss. As an estimate, we wrote a simple MPI implementation of the matrix scatter code in Figure 1 and Figure 2, running *on a single user thread* using the MPI library from IBM. This MPI code stores the input matrix a hash table at the threaded level, i.e. hash_insert_x, etc. This MPI code is also blocking; it uses only the standard blocking point-to-point communications MPI_Send and MPI_Recv. The test matrices are sparse matrices downloaded from the MatrixMarket website[1]. Table 1 shows the average barrier time in microseconds per non-zero values in the matrix. The two implementations ran using all four processors of four nodes of the IBM SP described later in Section 3.

Table 1 Averages of the barrier time of the matrix scatter over the number of non-zero matrix entries show that the relative cost of thread management decreases as the matrix becomes bigger.

Matrix	Size	Non-zeros	I/O Barrier	CSE_Hash Barrier	Send/Recv Barrier
s3dkq4m2	90449	2455670	543.2	61.1341	58.7964
e40r5000	17281	553956	2474.21	212.198	208.536
e40r0000	17281	553956	2478.91	210.793	209.767
af23560	23560	484256	2033.16	163.266	158.122
bcsstk17	10974	219812	273.357	41.7125	41.614
fidapm29	13668	186294	46.7143	6.84031	5.32564
bcsstk25	15439	133840	823.144	316.992	305.686
memplus	17758	126150	63.3752	16.1653	16.9388
bcsstk18	11948	80519	658.351	370.815	357.013
tols4000	4000	8784	39.7594	26.9704	18.3502
bwm2000	2000	7996	62.9534	31.4562	26.4441
west2021	2021	7353	69.9668	66.5721	53.1782
jpwh_991	991	6027	77.216	39.5261	27.4571
tols2000	2000	5184	29.2404	27.0639	7.02411
cdde6	961	4681	65.1188	23.7996	14.031
fs_541_4	541	4285	67.1141	26.5981	18.6019
impcol_d	425	1339	36.832	52.7199	14.5668
impcol_a	207	572	68.1591	45.6399	14.8374
impcol_c	137	411	77.4964	79.8248	9.08759
impcol_b	59	312	41.3205	53.6058	20.1987

There are other potentially more effective ways to guarantee atomic accesses to a hash table at the threaded level. One might split the lock, one might grab read-only locks whenever possible, or one might resort to lock-free mechanisms. Lock-free accesses and updates to table entries can be implemented using one-sided communications (get and store-conditional)[9,15]. Lock-free mechanisms avoid locking operations at the expense memory copies. The discarded memory must be reclaimed. The time to initiate an active message on the origin process is much bigger than the time for a mutex lock when the lock is ready, and so is the time to initiate a put operation on the target process. Since the values held in a table are atomic, accesses to the local table are quick compared to a mutex lock. Much of the time spent answering a remote hash table access is spent waiting to deallocate buffers after they are no longer needed. In the present context, it seems likely that items in the message queue would take longer to clear with lock-free accesses than with mutexes. A garbage collector could process obsolete copies, but might also hide the interactions between software parameters and wall-clock time(s). Note also that lock-free accesses and garbage collection would be justified if the values in the hash table were neither atomic nor small. An example might be that of a hash table of quadtrees.

3. Micro benchmarks

Three types of micro-benchmarks were defined, corresponding to different communication patterns, values in the table were single 32-bit integers and keys were themselves 32-bit integers. The same unsorted sequence of *m* unique keys was generated on each process; in the first set of benchmarks, shown in Figure 3, one million such keys were generated in the range from 0 to 7 millions. During each benchmark, three loops similar to that of Figure 2 are executed in sequence. Values are inserted in the first loop, searched for in the second and deleted in the last. The corresponding source code, not shown here, is very similar to that shown in Figure 2, except that the values and their keys are initially stored in two separate arrays rather than in a local hash table.

- In the best-case scenario *(1-N)*, process P_0 inserts, then looks up and finally deletes the whole set of key-value pairs from all *p* processes. All values for the key *k* are mapped to process number *k%p*, the remainder of the integer division of *k* by *p*. Runtime averages are taken over the number of values transferred to or from the *target* processes.
- The all-to-all scenario *(N-N)* is identical to the *1-N* scenario except for the fact that the i^{tch} process is both the target for all operations involving keys whose residue modulo *p* is *i* and the origin for all table accesses for the keys *m/p*i* through *m/p*i+i-1* in the sequence. Runtime averages are taken over the number of values inserted by each *origin* process.
- The worst-case scenario *(N-1)* is the opposite of the *1-N* scenario. Processes originate the same operations as in the N-N case but now process P_o is the target of them all. Runtime averages are taken over the number of values inserted by each *origin* process.

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. There was one process on each processor, so there were four processes on each SMP node. Measurements were done using the Performance Application Programming Interface (PAPI) library[2].

Figure 3 shows a comparison of the completion time of HASH_Insert and LAPI_Put for all three scenarios with a single value inserted with each key, in particular the communication request size *n* is one. There are some important difference between the benchmark loops for HASH_Insert and LAPI_Put. First, the LAPI loops store the values into an array of 7 millions elements using key values as indices. In this way, the addresses of the values on the target process are computed using simple arithmetic by the origin process. Second, the loop for timing LAPI_Put is unblocked: LAPI_Waitcntr is called immediately after LAPI_Put.

The worst-case completion times (N-1) increase linearly with the number of processors for both LAPI_Put and HASH_Insert. That is because only one thread is servicing all insertions and the total insertion time is effectively constant. Observe that the completion times for HASH_Insert in the N-N and 1-N scenarios are very close to that for the LAPI_Put 1-N case; that is also true for the initiation times.

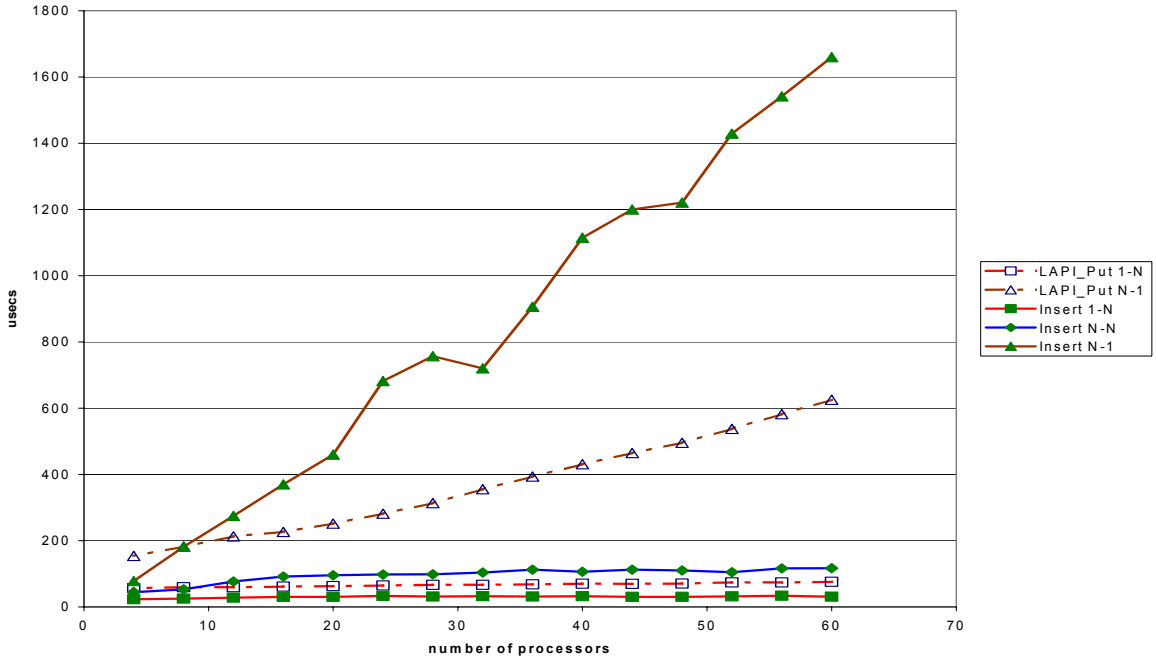

Figure 3 Comparison of the average completion times for PAPI_Put and HASH_Insert over the same sequence of random keys for best (1-N), average (N-N) and worst (N-1) case communication patterns.

4. Link between request and queue sizes

How robust are our results? The PAPI clock resolution on this IBM SP is about three microseconds; but in the context of inter-process communication, the variance of the run-time may be more informative. Therefore, we ran 200 N-N micro-benchmarks with random settings. Each randomized run used a distinct sequence of keys (ranging from 0 and 7 millions) by starting the random number generator with a distinct seed. The number of SP nodes varied between 1 and 11 with all four processors on each node active. The request size *n* for every table access varied randomly between one and eight. The blocking factor *L*, see Figure 2, varied randomly between 1 and multiples of 10 ranging from 10 to 200. The total number of keys was fixed at 10,000 times the number of processors. Figure 4 shows a scatter plot of the wall clock times divided by the number *n* of values transferred versus the blocking factor *L, with one data point (dash) for each process and each run*. Larger red dashes are for the immediate times, medium size blue dashes are for the completion times and the smaller brown dashes are for the barrier times. Wall-clock times for a single run tend to cluster together, leaving gaps within columns. The block length *L* appears to have little impact on the immediate wall-clock time. However, the variance of the completion wall-clock time seems to decrease when *L* increases. Only four of the runs had a value of L equal to 130, the corresponding column is clearly shorter than the all other columns.

The greatest part of the wall-clock time is spent, by far, in completion handlers waiting to free buffer space that is no longer needed[13]. When, for instance, *n* values for key *k* are removed from process P_t by process P_o, the completion handler on P_t first allocates a buffer *b* to hold the values it will remove from the local table. This buffer is then passed on to the LAPI_Put operation that will transfer the values back to P_o. Clearly, *b* cannot be released before the call to LAPI_Put completes locally. As was described earlier, such active transactions are placed in a queue of size *q* to await release of their buffers. The queue size in Figure 4 is 128. No such reduction is apparent in a similar graph using the same micro-benchmark parameters but a queue size of zero, which is when each completion handler frees all the storage allocated by the corresponding header handler. Its therefore seems unlikely that such a reduction would be due to sampling only. The analysis that follows confirms this impression. This analysis is based on the Kruskal-Wallis test for equality of distributions among independent data series.

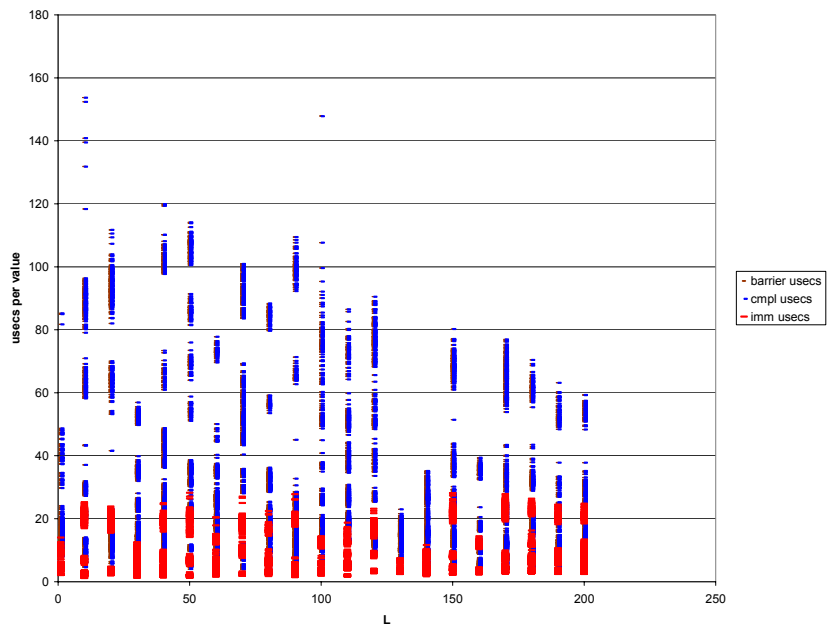

Figure 4 Blocking is seen to reduce the variance of the wall clock time over 200 random micro-benchmarks with insertion, search and deletion on 4 to 44 processors of an 4-Way 375MHz IBM SP.

The Kruskal-Wallis test is a well-known non-parametric (rank) test for testing equality of distributions among *r* independent data series. Let X_{ij} denote the ith sample of the jth series with $m_j \geq i \geq 1$ and $m_j \geq 5$ for all j. If $y_1, \ldots y_m$ is the sorted sequence of all observed values, including repetitions, then the rank R_{ij} of a sample X_{ij} is the average of all positions k with $y_k = X_{ij}$. For instance, if the 4 observed values were 0.001, 0.2, 0.1 and 0.2, the computed ranks would be 1, 3.5, 2 and 3.5 respectively. One defines a statistic H whose distribution is approximately (asymptotically) a χ^2 distribution with *r*-1 degrees of freedom. For completeness sake, H is defined by the equation below where μ_j denotes the mean rank for the jth series and μ is the mean rank over all series.

$$\mu_j = \tfrac{1}{m_j} \sum_{i=1}^{m_j} R_{ij}$$

$$\mu = \tfrac{1}{m} \sum_{j=1}^{r} m_j \mu_j$$

$$H = \frac{12 \sum_{j=1}^{r} m_j \left(\mu_j - \mu\right)^2}{m + m^2}$$

The Kruskal-Wallis test rejects the null hypothesis that the samples within the series are identically distributed when the computed value of H exceeds an appropriate threshold. For example with 20 series, one would reject the null hypothesis at the 0.05 significance level whenever the computed value of H is greater than or equal to 30.14. Further details about the H statistic can be found in any good textbook on sturdy statistics. In general, each data series should have at least five samples for the test to be meaningful. In addition, there should not be too many ties among samples and those samples must be faithful to the data. The last point is easily overlooked in the present context. Wall-clock times are recorded in terms of completed CPU cycles and converted in microseconds. They are stored as 32-bit variables and printed with 6 digits of precision. Not all those digits are meaningful and different conclusion may be reached depending on how many digits one considers when computing H.

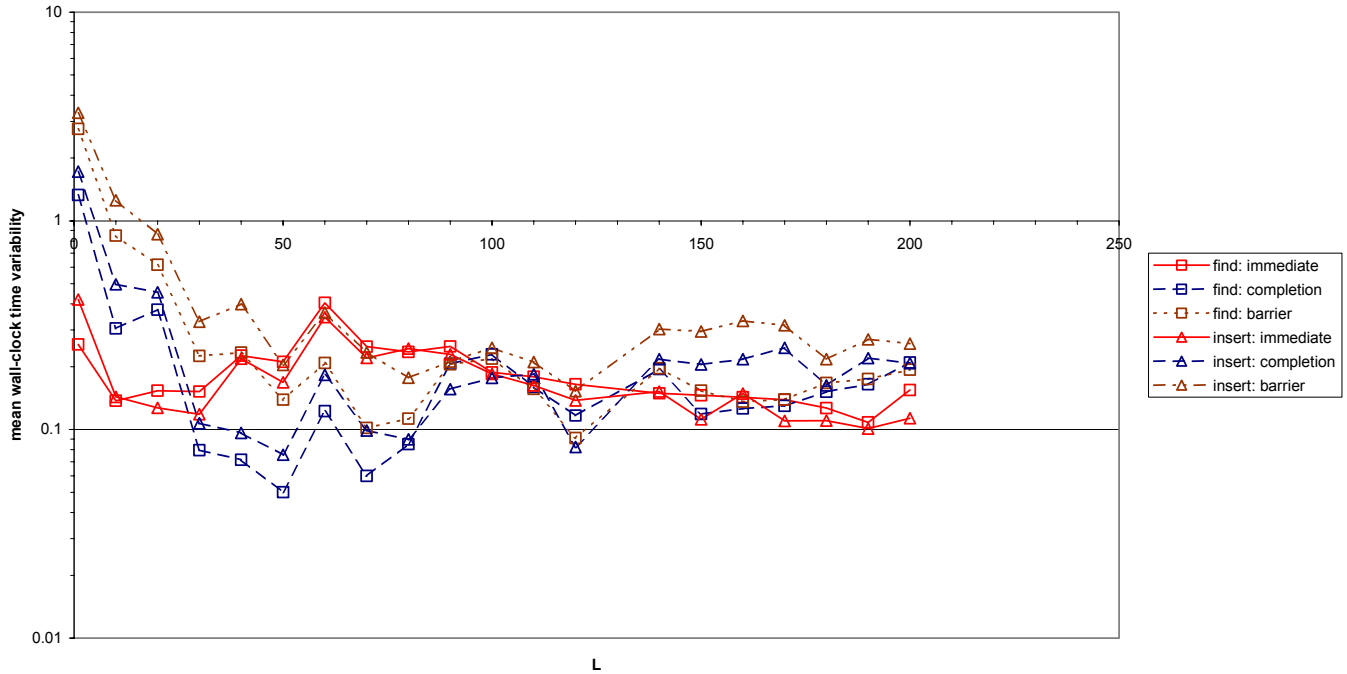

Figure 5 Block sizes L near half the queue size $q=128$ are seen to reduce the relative difference between the mean and maximum completion and barrier wall-clock times.

The data records being dependent it is convenient to define a measure of variability in the wall-clock times for each run. The wall-clock time *variability* is the ratio whose numerator is the difference between the largest wall-clock time and mean wall-clock time and whose denominator is the mean wall-clock time. This gives us a single number for each run, actually one number for each of the three types of wall-clock times. Those numbers can be regarded as both commensurate and independent of one another. Figure 5 shows the mean variability as a function of the blocking factor L for insertions and searches. Wall-clock times for deletions are similar to those for searches and are not shown. The wall-clock times for each of the six series were analyzed separately. Each of the resulting datasets consists of 21 sequences one for each value of L. Suppose that the blocking factor L had no effect on the wall-clock time; one would expect that the distribution within each of the 21 subsets would be the same. When the Kruskal-Wallis test rejects the above null hypothesis at a 0.05 level of significance, there is a 5% chance that the distribution of wall-clock times does not vary across subgroups, that is that L has no impact. This test does not elucidate the nature of any variability. Among all 21 series, only one (for L = 130) had fewer than 4 points and it was excluded from the analysis. We also truncated the raw variability data to one digit after the decimal point. The resulting H statistics are shown in Table 2 both for a queue size q=0 and q=128. The blocking factor appears to loose its relevance when the queue is disabled.

Table 2 Kruskal-Wallis H statistics for the wall clock-time versus the blocking factor *L*, suggest a relationship between effective values of *L* and the queue size *q*. Values of H leading to rejection of the null hypothesis at the 0.05 significance level are shown in bold.

q	Find Immediate	Find Completion	Find Barrier	Insert Immediate	Insert Completion	Insert barrier
0	12.4827	16.849	30.2172	**81.9073**	**71.2191**	**61.5092**
128	**72.743**	**58.6451**	**50.6353**	**80.6566**	**70.5592**	**55.835**

An analysis of the hardware event counts generated by PAPI shows that the total number of elapsed cycles is less closely related to cycles stalled waiting for memory accesses in runs with L equal to 50 than in runs with L equal to 1, 10 or 60. A plausible explanation is that when the blocking factor L is small the load among processes is not as uniformly distributed as when L is large. On the other hand, when L is large compare to the queue length q, it becomes more likely for a completion handler to will have to wait before being able to free buffers from a previous transaction.

5. Scalability

Figure 6 shows a scatter plot of all-clock time versus number of processors for the same data as in Figure 4. Again, each dash represents one wall-clock time for one process during one of the 200 randomized micro-benchmarks. The same color and size code is used as well. The peak at 16 processors and the drop at 44 are a reflection of the sampling.

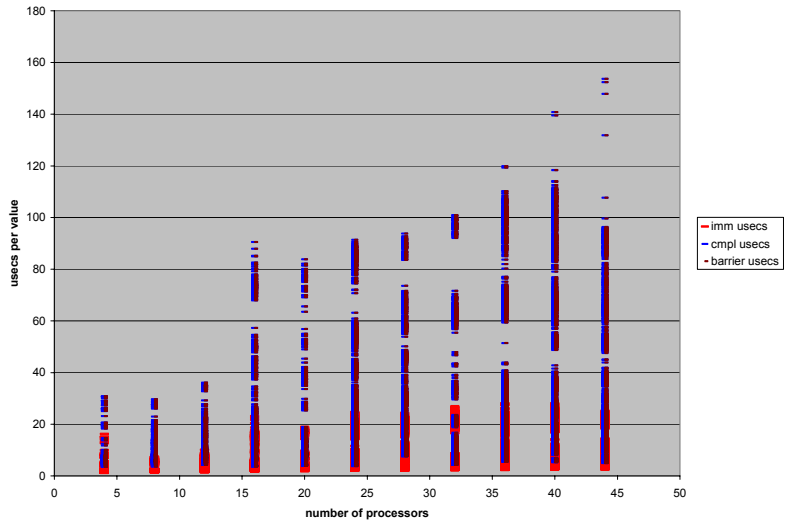

Figure 6 Scalability of insertions, searches and deletions over 200 random micro-benchmarks with insertion, search and deletion on 4 to 44 processors of an 4-Way 375MHz IBM SP.

One would expect that for a scalable code, the number of processors should not have a significant impact on the variability of the wall-clock time as defined in the previous section. A Kruskal-Wallis test may be used for showing, possibly with a high degree of confidence, that a parallel code is not scalable. In particular, this test is not strong enough to demonstrate scalability in this way. Here, we partitioned the data into 11 groups according to the number of processors. The series for 44 processors was excluded for not having enough samples. Table 3 shows the computed H statistic with and without the message queue enabled. The figures that appear in parenthesis are the values of H when the group for 4 processors is excluded. The reason this series has such a strong impact is that communication within a node is significantly faster than across nodes.

Table 3 Kruskal-Wallis H statistics, for the wall-clock time versus the number of processors, illustrate the importance of the message queue with respect to scalability. Values leading to reject the null-hypothesis at the 0.05 significance level are shown in bold.

q	Find Immediate	Find Completion	Find Barrier	Insert Immediate	Insert Completion	Insert Barrier
0	**36.21** (8.74)	14.32 (9.02)	**42.67** (13.98)	**25.69** (10.15)	**27.03** (4.53)	**87.82** (**19.89**)
128	10.19 (11.11)	16.21 (6.59)	**23.17** (**29.72**)	10.43 (12.07)	9.49 (**21.78**)	**36.68** (**64.48**)

6. Conclusion

A distributed hash table library implemented on top of active messages has been presented. The effectiveness and scalability of this library were shown to be both robust and comparable to that of one-sided LAPI operations. The existence of a blocking factor affecting the variance of wall-clock time was demonstrated experimentally using randomized micro-benchmarks. The size and nature of performance data sets offer some interesting problems in data analysis, that were only alluded to in this paper. Finally, we demonstrated how a one-sided communication for linear algebra operations, the matrix scatter, can be implemented using distributed hash tables.

Acknowledgements

We would like to thank Jesper Träff, Dave Cowley and the anonymous reviewers for their comments, help and suggestions. This research was performed in part using the Molecular Science Computing Facility (MSCF) in the William R. Wiley Environmental Molecular Sciences Laboratory at the Pacific Northwest National Laboratory. The MSCF is funded by the Office of Biological and Environmental Research in the U.S. Department of Energy. Pacific Northwest is operated by Battelle for the U.S. Department of Energy under Contract DE-AC06-76RLO 1830. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract Number DE-AC03-76SF00098.

Bibliography

1. Boisvert R., Pozo R., Remington K., Barrett R., and Dongarra J., "Matrix Market: a web resource for test matrix collections", in *The Quality of Numerical Software: Assessment and Enhancement,* R. Boisvert, ed., Chapman and Hall, London, 125-137, 1997.
2. Browne S., Dongarra J. J., Garner N., Ho G., and Mucci P, "A Portable Programming Interface for Performance Evaluation on Modern Processors", The International Journal of High Performance Computing Applications 14[3], 189-204, 2002.
3. Carissimi A. and Pasin M., "Athapascan: An experience on Mixing MPI Communications and Threads", in 5th European PVM/MPI Users' Group Meeting, Alexandreov V. and Dongarra J. J. , eds., Springer Verlag Lecture Notes in Computer Science 1497, 137-144, 1998.
4. Carriero N. and Gelernter D., "LINDA in context", Communications of the ACM, 32[4], 444-458, 1989.
5. Chiola G. and Ciaccio G., "Architectural issues and preliminary benchmarking of a lowcot network of workstations based on active messages", in 14th ITG/GI Conference on Architecture of Computer Systems, 13-22. 1997.
6. Ciardo G., Gluckman J., and Nicol D., "Distributed State-Space Generation of Discrete-State Stochastic Models", Institute for Computer Applications in Science and Engineering, NASA CR-198233 ICASE Report No. 95-75. 1995.
7. Codd E. F., "A relational model for large shared databanks", Communications of the ACM, 13[6], 377-387, 1970.
8. Geist A., Gropp W., Huss-Lederman S., Lumsdaine A., Lusk E., Saphir W., Skjellum T., and Snir M., "MPI-2: Extending the message-passing interface", Argonne National Laboratory, MCS-P568-0296, 1996.
9. Herlihy M., "A methodology for implementing highly concurrent data objects", ACM Transactions on Programming Languages and Systems, 15[5], 745-770, 1993.
10. High Performance Fortran Forum, "High Performance Fortran Language Specification, version 1.0", CRPC-TR92225. 1993.
11. Joshi M., Karypis G., and Kumar V., "ScalParC: A New Scalable and Efficient Parallel Classification Algorithm for Mining Large Datasets", in 11th International Parallel Processing Symposium (IPPS), 1998.
12. Knuth D.E. "Linear Probing and Graphs", Algorithmica, 22[4], 561-568, 1998.
13. Malard J.M. and Stewart R.D., "Hidden cost of memory management in asynchronous communication: A Distributed Hash Table Example", poster at Scaling to New Heights Workshop, 2002.
14. Martin R.P., "HPAM: An active message layer for a network of HP workstations", University of California at Berkeley , Technical Report CSD-96-891, 1996.
15. McKenney P.E., "Selecting locking primitives for parallel programming", Communications of the ACM, 39[10], 75-82, 1996.
16. Miller K., "Moving Finite Element II", SIAM J.Numerical Analysis, 18, 1033-1057, 1981.
17. Miller K. and Miller R. N., "Moving Finite Element I", SIAM J.Numerical Analysis, 18, 1019-1032, 1981.
18. Nichols B., Buttlar D., Proulx-Farrell J., and Farrell F., *Pthreads Programming: A POSIX Standard for Better Multiprocessing*, O'Reilly, 1996.
19. Nieplocha J. and Carpenter B., "ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems", in Springer-Verlag. Lecture Notes in Computer Science 1586, 1999.
20. Nieplocha J., Harrison R.J., and Littlefield R.J., "Global arrays: a portable programming model for distributed memory computers", in Supercomputing'94, 340-349, 1994.
21. Parashar M. and Browne J.C., "System Engineering for High Performance Computing Software: The HDDA/DAGH Infrastructure for Implementation of Parallel Structured Adaptive Mesh Refinement.", in IMA Volume 117: Structured Adaptive Mesh Refinement (SAMR) Grid Methods, 1-18, 2000.
22. Parashar M., Browne J.C., Edwards C., and Kimkowsky K., "A Common Data Management Infrastructure for Parallel Adaptive Algorithms for PDE Solutions", in Supercomputing '97, 1997.
23. Schloegel K., Karypis G. and Kumar V., "A Unified Algorithm for Load-balancing Adaptive Scientific Simulations", in Supercomputing 2000.
24. Shah G., Nieplocha J., Mirza C., Harrison R., Govindaraju R.K., Gildea K., DiNicola P. and Bender C., "Performance and experience with LAPI: a new high-performance communication library for the IBM RS/6000 SP", in International Parallel Processing Symposium, 260-266, 1998.
25. Shan H., Singh J.P., Oliker L. and Biswas R., "A Comparison of Three Programming Models for Adaptive Applications on the Origin2000", in Supercomputing 2000.
26. Stagg A.K., Hallberg J.P., and Schmidt J.H., "A Parallel, Adaptive Refinement Scheme for Tetrahedral and Triangular Grids", in IPDPS Workshops 2000, 512-518, 2000.
27. Standish T.A., *Data Structure techniques*, Adison-Wesley, 1980.
28. Warren M.S. and Salmon J.K., "A parallel hashed Oct-Tree N-body algorithm", Supercomputing 93, 12-21. 1993.
29. Yelick K., Chakrabarti S., Deprit E., Jones J., Krishnamurthy A., and Wen C., "Parallel Data Structures for Symbolic Computation", in Workshop on Parallel Symbolic Languages and Systems, 1995.
30. Zuberek W.M. and Rada I., "Distributed generation of state space for timed Petri Nets", in High Performance Computing Symposium 2001 (HPC'01), 219-227, 2001.