

Hashing Strategies for the Cray XMT

Eric L. Goodman*, David J. Haglin†, Chad Scherrer†,
Daniel Chavarría-Miranda†, Jace Mogill†, John Feo†

*Sandia National Laboratories
Albuquerque, NM, 87123 USA
elgoodm@sandia.gov

†Pacific Northwest National Laboratory
Richland, WA, 99354 USA
{david.haglin, chad.scherrer, daniel.chavarría, jace.mogill, john.feo}@pnl.gov

Abstract—Two of the most commonly used hashing strategies—linear probing and hashing with chaining—are adapted for efficient execution on a Cray XMT. These strategies are designed to minimize memory contention. Datasets that follow a power law distribution cause significant performance challenges to shared memory parallel hashing implementations. Experimental results show good scalability up to 128 processors on two power law datasets with different data types: integer and string. These implementations can be used in a wide range of applications.

I. INTRODUCTION

Hash tables are a fundamental data structure used throughout many application domains. Hash tables can be applied to problems that require efficiently mapping a set of data elements (typically called *keys*) into another set of data elements (typically called *values*). Other names for the abstract notion of mapping keys to values are *associative array* and *dictionary*. Many data structures can be used for this problem, but hash tables have been found to be able to provide “constant” time lookups and insertions into the structure.

Hash tables utilize a *hash function* that maps keys to indices of an array called the *hash table* (the entry in the table is typically called a *bucket*), where the key and its corresponding value resides. The ideal hash function is one that uniquely maps a key into a bucket, however that is rarely achieved in practice, so a strategy to deal with *collisions*—multiple keys mapping to the same bucket—must be an integral part of the definition and implementation of the hash table. Two common strategies for dealing with collisions are: *linear probing* (sometimes called linear open addressing), where the location of the second and later keys with the same hash function value are stored in the region of the hash table (array) immediately following the initial slot; and *hashing with chaining*, where instead of the hash table containing the keys and values, it contains head pointers to linked lists containing the keys and values. Given a reasonable hash function that exhibits a low collision rate, the average time cost of looking up a key in the hash

table tends to be **constant**, independent of the collision handling mechanism used.

Variations on the use of hash tables allow for a wider range of applications with only slight changes to the infrastructure. Rather than assigning a value to a key (as in a dictionary), it is possible to count the number of times a key is encountered. This variation can be used to count the frequency of occurrences of keys; a typical example of this is to count the frequency of words in a corpus of text documents. We call this variation *hash-based frequency counting*.

Hash tables have been used for a long time in computing and in a wide range of applications. An entire chapter in the popular text by Cormen, *et al.*, is devoted to the use of hash tables [1]. According to Donald Knuth [2], H. P. Luhn was the first to describe hashing techniques in an internal IBM memo in 1953. In spite of the pervasiveness of hashing strategies, practical techniques for implementing hash tables on shared-memory parallel machines have not been explored in depth. Moreover, several applications developed for the XMT utilize a hashing strategy as part of their application [3]–[5].

A. Background of Parallel Hashing

Hashing strategies have been used extensively in a wide range of computing applications. Running these applications on parallel machines faces a particular set of challenges to achieve good performance; most notable is the challenge of memory contention. In their 1986 work on parallel hashing, Karlin and Upfal explore strategies to minimize memory contention on a PRAM and proposed an algorithm for p processors to store and retrieve an arbitrary set of p data items in $O(\log p)$ parallel steps [6]. Lee explored parallel hashing on *bulk synchronous parallel* (BSP) and *queued shared memory* (QSM) models of computation [7]. Distributed hash tables are used in a wide range of popular applications, including Domain Name System (DNS), peer-to-peer file sharing, and content distribution such as BitTorrent.

B. Dynamic Memory in Hashing with Chaining

One of the challenges that must be addressed for an efficient implementation of hashing with chaining is the mechanism used to allocate memory when adding a new entry in to a chain. We have developed a mechanism that is an amalgamation of two specialized memory pool techniques: *arena memory management* [8] and *region-based memory pools* [9]–[11].

Arena memory management is a dynamic memory strategy that uses a pool of large buffers, where the locking is done on each of the buffers. Thus, for multithreaded access to this memory pool, creating an *arena* for each thread eliminates the need for locking that exists when multiple threads share a single arena. The cost for this approach is the extra memory needed to have a separate arena for each thread. This can work very well for small to modest numbers of threads.

Region-based memory pools consists of large buffers (*regions*) from which smaller buffers are allocation and the *free* operation is done once for either all of the N most recent allocated smaller buffers or all of the buffers in the region. This concept usually also includes the notion of putting all instances of specific types into regions [10]. This technique was originally developed to support functional languages and the feature of *type-safety* in addition to efficient, stack-like activation records.

Our work on the hashing with chaining strategy uses parts of arena and region-based memory management systems. We minimize the locking imposed by the memory pool (a goal of arena memory management systems) by localizing thread synchronization needed to allocate a small buffer to one atomic operation. We also utilize the properties of the region-based memory management system that supports the notion of quickly placing many small buffers back in the global memory pool by simply doing a free on each of the regions rather than freeing each small buffer individually.

C. Cray XMT

The Cray XMT is the commercial name for the new shared-memory multithreaded machine developed by Cray under the code name “Eldorado” [12], [13]. The system is composed of dual-socket Opteron AMD service nodes and custom-designed multithreaded compute nodes with *Threadstorm* processors. The entire system is connected using the Cray Seastar-2.2 high speed interconnect. The system we use in this study has 128 processors and 1 TB of shared memory.

Each Threadstorm processor is able to schedule 128 fine-grained hardware threads (the XMT terminology for this is *stream*) to avoid memory-access generated pipeline stalls on a cycle-by-cycle basis. At runtime, a software thread is mapped to a hardware stream comprised of a program counter, a status word, 8 target

registers and 32 general purpose registers. Each Threadstorm processor has a VLIW (Very Long Instruction Word) pipeline containing operations for the Memory functional unit, the Arithmetic unit and the Control unit.

Memory is structured with full-empty-, pointer forwarding- and trap- bits to support fine grained thread synchronization with little overhead. The memory is hashed at a granularity of 64 bytes and fully accessible through load/store operations to any Threadstorm processor connected to the Seastar-2.2 network, which is configured in a 3D toroidal topology.

The software environment on the Cray XMT includes a custom, multithreaded operating system for the Threadstorm, a parallelizing C/C++ cross-compiler targeting Threadstorm, a standard Linux 64-bit environment executing on the service and I/O nodes, and the necessary libraries to provide communication and interaction between the two parts of the XMT system. The parallelizing compiler generates multithreaded code that is mapped to the threaded capabilities of the processors automatically. Parallelism discovery happens fully- or semi-automatically by the addition of `pragmas` (directives) to the C/C++ source code. This discovery focuses on analyzing loop nests and mapping the loop’s iterations in a data-parallel manner to threads.

To understand the lightweight synchronization features of the XMT, we review two aspects of the programming model: full-empty bits and generic functions. Each 8-byte word of memory has an associated full-empty bit enabling lightweight synchronization operations. The software (compiler and runtime) allows programs to manipulate the full-empty bits with generic functions are executed atomically within one instruction cycle. Our code uses these generic functions:

- *readxx*: Returns the value of a variable without checking the full-empty bit.
- *readfe*: Returns the value of a variable when the variable is in a full state, and simultaneously sets the bit to be empty.
- *writfef*: Writes a value to a variable if the variable is in the empty state, and simultaneously sets the bit to be full.
- *int_fetch_add*: Atomically adds an integer value to a variable.

D. XMT-Specific Issues For Hashing

The foremost concern with hashing on the XMT is the memory contention. There are two types of contention that needs to be considered: synchronization and hotspotting. The synchronization issues are similar to those of the general PRAM model and involve locking (or other synchronization mechanisms) to ensure correctness of the data structure while multiple threads attempt to alter the underlying data structure.

Multiple threads attempting to read the same memory location at the same time will serialize and cause a *read hotspot*. Some of the threads may be delayed long enough to be captured by a *long latency trap*. The processing of these long latency traps can cause a significant (10-fold is not uncommon) increase in processing time. As an example, consider the situation where 10,000 threads are trying to look up something in a hash table. Although the 10,000 threads may be seeking 10,000 different slots (or buckets), they must all read the pointer to the hash table and compute the offset into the hash table to find their slot. To avoid the read hotspot on the hash table pointer, the programmer (or the compiler) must be aware of whether each thread loads the hash table pointer into a register, thus avoiding the read hotspot.

E. Two Hashing Strategies Under Study

Hashing strategies are categorized by their mechanism for dealing with *collisions* (when two different keys map to the same hash value). One of the first hashing strategies used was linear probing [1]. This technique has the limitation that the hash table must be at least as big as the number of keys to be inserted. A common technique for hashing with an unbounded key capacity is *hashing with chaining*. Each of the hash table entries are really head pointers to a linked list of nodes holding the keys (and their associated values) that all hash to the same *bucket*.

F. Our contributions

We investigated the performance of two of the most commonly used hashing strategies under several kinds of input data that are particularly challenging for hashing mechanisms on a multithreaded architecture. Our implementations were tailored specifically for the Cray XMT multithreaded system and show good scalability. For the hashing with chaining strategy, we propose a new data structure to support fast allocation for our linked lists in the hashing with chaining strategy. This mechanism is called *Hashing with Chaining and Region-based Memory Allocation* (HACHAR).

The rest of the paper is organized as follows: section II presents details of our two implementations; section III describes our datasets used and presents our experimental results on those datasets; and section IV provides our assessment of the experiments and suggests future work.

II. IMPLEMENTATIONS

All our implementations are of the hash-based frequency counting variation since a significant factor leading us to investigate hashing strategies was the application of counting word frequencies in a corpus of text documents.

A. Common Hash-table Operations

Our implementations provide common operations: data structure initialization (constructor), search, insert, and data structure tear-down (destructor). In our target applications the remove (or delete) operation is not needed, so we did not include this operation in our implementation. We expect that including a remove operation would not alter the scalability.

1) *Searching for a Key*: In all our implementations the task of searching for a key (without inserting) can be done without any thread synchronization. For linear probing, we compute the hash function to get the initial array location, then proceed to step forward through the array looking for either the desired key or an empty slot. Note that “stepping forward” may involve cycling around from the bottom to the top of the array. If an empty slot is encountered, we return a value indicating “not found”. Otherwise we return the value associated with the key. For chaining, we compute the hash function to get the bucket location, then proceed to step through the linked list of entries in that bucket looking for the desired key.

2) *Inserting a Key*: When inserting a key, there are two situations that may occur: (i) the key already exists in the hash structure, in which case we *update* the value by incrementing the associated count; or (ii) the key does not yet exist in the hash structure, in which case we need to (carefully) insert the key. This careful insertion is where synchronization must occur, and we must *acquire a location*. The algorithm used to acquire a location is one of the fundamental tasks in all our implementations of hashing and is very similar across all of our implementations. At a high level, this is a two-step process of first looking without locking followed by a lock (if necessary). As shown in Figure 1, we first look at the location to see if it is empty (if not, the attempt to acquire the location fails). Since some other thread may be trying to acquire the same location, we must lock the location and look again to see if it is still empty. If the location is still empty after locking, we have acquired the location, so we mark it as reserved for this new key. If we do lock the location, we must always unlock when done processing the request.

3) *Removing a Key*: Although many applications do require support for removing a key, we did not include that in our study.

B. Linear Probing

Our linear probing implementation utilizes three, identically sized arrays: a key, value, and occupied array. The key/value combination for a particular index in the array can be thought of as a slot in the hash table, and the corresponding element in the occupied array is a boolean, signifying whether or not the slot has been claimed. Corresponding elements from the three arrays are passed

Procedure: TwoStepAcquireAttempt
Parameters: (*location, key*)

```

// first check without locking
1: if location is empty then
2:   lock(location)
   // then check with the lock
3:   if location is still empty then
4:     Reserve location for key
5:     unlock(location)
6:     return true
7:   end if
   // Another thread modified the structure before
   // we could acquire location
8:   unlock(location)
9: end if
10: return false ▷ Caller needs to continue searching

```

Fig. 1: General procedure to acquire some memory location. The caller may need to continue searching for the *key* (after being returned a value of *false*) due to other threads that altered the structure before the caller was able to acquire the location.

as parameters to the InsertLinearProbing procedure in Figure 2. The outcome from calling InsertLinearProbing is one of the following:

- If the slot is already holding a key (the slot key) and it matches the candidate key, we consider the slot claimed.
- If the slot is empty we can do the two step acquire successfully, we fill in the slot with our candidate key and consider the slot claimed.
- If, however, the slot is already holding a key that does not match our candidate key, we return false indicating that the current slot is not available for this candidate key.

C. Chaining with Region-based Memory Allocator

Recall that hashing with chaining involves hashing the key to get the bucket index followed by chaining forward in a linked list in search of the key. If the key is found, the appropriate value is returned. If key is not found, we either insert a new key into the chain or return an indication that the key is not in the hash structure.

Although there is a good, general purpose memory manager on the XMT designed specifically for multi-threaded architectures [14], our memory allocation pattern is well-known and specialized, so it makes sense to implement a region-based memory allocator to achieve better performance. We also believe that allocating many small buffers, all of the same size, from a mechanism designed as a general purpose memory manager is susceptible to memory contention issues. Our region-based memory allocator provides for all linked list nodes in the buckets to be allocated out of large regions, structured as shown in the “Hash Table” of Figure 3.

Procedure: InsertLinearProbing
Parameters: (*lock, slotKey, candidateKey*)

```

1: status = readxx(lock)
2: if status == claimed then
3:   if slotKey == candidateKey then
4:     return true
5:   end if
6: else
7:   status = readfe(lock)
   // The first thread to reach the slot?
8:   if status == unclaimed then
9:     slotKey = candidateKey
10:    writeef(lock, claimed)
11:    return true
12:   else
13:     if slotKey == candidateKey then
14:       writeef(lock, claimed)
15:       return true
16:     end if
17:     writeef(lock, claimed)
18:   end if
19:   return false
20: end if

```

Fig. 2: Procedure to insert (or update) a key-value pair in a linear probing structure. This procedure returns *true* to indicate the slot is holding the key we are searching for. A *false* return value indicates that the slot is holding a key different from the one we are search for. Note that the two step acquire procedure is represented in lines 2, 7–12, and 17–19.

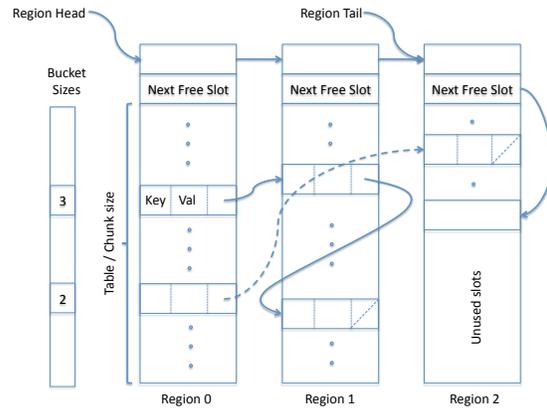


Fig. 3: Data structure used to support the Hashing with CHaining And Region-based memory allocation(HACHAR).

The data structure is initialized by allocating a region to serve as the hash table, allocating a bucket sizes array, and writing zeroes in the bucket sizes array. In anticipating buckets that exceed a size of one, an empty region is allocated and linked from the hash table. To dismantle this data structure at the end of using the hash table all that is needed is to free the bucket sizes integer array and each of the regions. If the data type of the key or value needs to be dismantled, then each of the array entries will need to be freed as well.

Procedure: InsertHACHAR(*key*)

```

1: bucket = hashFunction(key)
2: if InsertIntoEmptyBucket(key, value = 1, bucket)
   then
3:   return
4: end if
5: while forever do
6:   Walk down linked list looking for key
7:   if (key is found) then
8:     int_fetch_add( value associated with key )
9:     return
10:  end if
11:   // at end of list, try to insert into the list
12:   pointer = readfe(currentNode.Next)
13:   // still at end of list after locking?
14:   if pointer == NULL then
15:     newNode = Allocate(key, value = 1)
16:     writfe(currentNode.Next, newNode)
17:     return
18:   end if
19:   // Some other thread modified list, we need
20:   // to continue searching from current location
21:   writfe(currentNode.Next, pointer)
22: end while

```

Fig. 4: Procedure to insert (or update) a key-value pair in a HACHAR structure. Note that the two-step acquire operation is embedded here in lines 11-17.

1) *Acquiring a Linked List Node Location:* If the key is not already there during an insert operation, a new node will need to be inserted. The challenge with this operation is to ensure that exactly one thread will insert the new node. To achieve this, we need to find the end of the linked list, lock out other threads, check to make sure we are still at the end of the linked list, then allocate a new node from the unfilled region, and finally link it in. Note that this process follows the two step acquire described in Figure 1. The locking is done by using the full-empty bit of the linked list *next* pointer itself. The rare occasion when the unfilled region is actually full will be discussed separately. The insert method is given in Figure 4.

2) *Memory Contention and Thread Synchronization:* All of this infrastructure is set up for the purpose of minimizing memory contention and synchronization issues. We have achieved a nearly lock-free search process where many threads can follow a linked list looking for their key. If many threads find the same key, then the “update” procedure requires only the synchronization handled by the atomic *int_fetch_add* operation.

The insert procedure is more involved. If the key already exists, then the synchronization is essentially equivalent to the search operation. If there are an abundance of unused slots in the unfilled

Procedure: InsertIntoEmptyBucket**Parameters:** (*key*, *value*, *bucket*)

```

1: if bucketSize[bucket] == 0 then
   // Lock out other threads
2:   status = readfe(bucketSize[bucket])
3:   if status == 0 then ▷ Check again with lock
4:     Set bucket’s head node to: key = value
5:     writfe(bucketSize[bucket], 1)
6:     return true
7:   end if
   // Some other thread modified list, we need
   // to go back and insert into non-empty bucket
8:   writfe(bucketSize[bucket], status)
9: end if
10: return false

```

Fig. 5: Procedure to insert (or update) into an empty bucket in a HACHAR structure. For data with lots of repetitions, this procedure will merely fail the if at line 1 and then return false indicating that nothing was done.

Procedure: Allocate(*key*, *value*)

```

// load a thread-local copy of tail pointer
1: ptr = tail
2: idx = int_fetch_add(ptr→next_free_slot, 1)
   // If tail region is full, allocate a new region
3: while index ≥ tableSize do
4:   oldTail = readfe(tail)
   Is tail region still full (after locking)?
5:   if oldTail == ptr then
6:     oldTail = new RegionBuffer
7:     ptr→next = oldTail
8:   end if
9:   writfe(tail, tail→next)
10:  ptr = tail
11:  idx = int_fetch_add(ptr→next_free_slot, 1)
12: end while
13: Set slot ptr→array[idx] to: key = value
14: return &ptr→array[idx]

```

Fig. 6: Procedure to allocate a new linked list node from the unfilled region in a HACHAR structure. Note that with many threads hitting this procedure at nearly the same time, the *next_free_slot* pointer will move well beyond the end of the current region. This works since we only check for ≥ to the region size and conclude that we must move to the next region (allocating one as we go unless some other thread allocates a region for us).

region, then the only synchronized used is the *int_fetch_add*(*next_free_slot*, 1) in the unfilled region. This atomic operation gives each thread a unique index thereby allocation a specific linked list node to each thread. When this region becomes full, several threads may end up an index pointing beyond the end of the region’s array. All of the threads in this situation must then coordinate which thread will allocate a new region and all of the threads must move over to this

new region and allocate a linked list node from there. To minimize the amount of time these threads must wait for the one thread to allocate a new region, it is reasonable to have an “on deck” region pre-allocated so that the new (empty) region can be simply linked in when needed. Using this scheme, a new “on deck” region would need to be allocated at some other time. For example, when a thread is allocated a linked list node and the unfilled region is exactly half full, this thread could be redirected to allocate an “on deck” region.

III. EXPERIMENTS

In order to test our hashing strategies on the XMT, datasets that stress the performance issues of the XMT are required. Datasets containing a few keys with very high frequency counts will render any hash table implementation susceptible to read hotspots in the hash buckets holding the highly-frequent keys. Datasets that follow a power law distribution exhibit this pattern.

We examined the performance of the two strategies using three data sets, a set of uniformly distributed integers, a set of integers constructed to follow Zipf’s law, and a snapshot of the english Wikipedia documents. For convenience, we will refer to the total number of elements in each data set as S_x and the number of unique elements as U_x where $x \in \{\text{uniform, Zipfs, Wikipedia}\}$. We use S_{array} to denote the size of the hash table. Of the three datasets used in this study, two have the power law distribution property. For comparison, we included the uniformly distributed integers dataset since it does not have this property.

All of these experiments were run on a 128 processor Cray XMT, possessing a total of one terabyte of memory. Also, the times we report in the experiments below are the time to insert the data into the array. We do not include times to load the data from disk, or construct or remove data structures.

A. Integers, Uniform Distribution

We generated a set of uniformly distributed integers in the range $[-2^{63}, 2^{63} - 1]$ where $S_{\text{uniform}} = 5 \times 10^9$ and $U_{\text{uniform}} = 5 \times 10^9$. The primary purpose of this experiment is to test each strategy’s ability to handle collisions. We chose the size of 5 billion integers to be similar in size to the Zipfian data set (described later in Subsection III-B). For linear probing, we vary S_{array} from $\frac{S_{\text{uniform}}}{0.95}$ to $\frac{S_{\text{uniform}}}{0.2}$, so that the load factor is $0.2 \leq l \leq 0.95$. Since chaining is dynamic and can handle when $U_x > S_{\text{array}}$, we also tried l up to 2. The hash function we use is simple, the key multiplied by a large, prime constant:

$$\text{hash}(x) = x \cdot 31,280,644,937,747 \quad (1)$$

To convert the hash into a location in the array, we performed the following operation

$$\text{getArrayIndex}(x) = \text{hash}(x) \& (2^{53} - 1)\%S_{\text{array}} \quad (2)$$

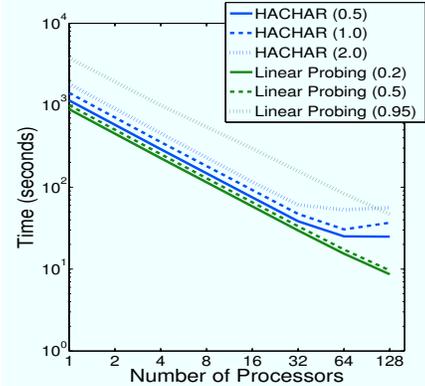


Fig. 8: Comparison between Linear Probing and HACHAR for different load factors on the uniform random integer data set.

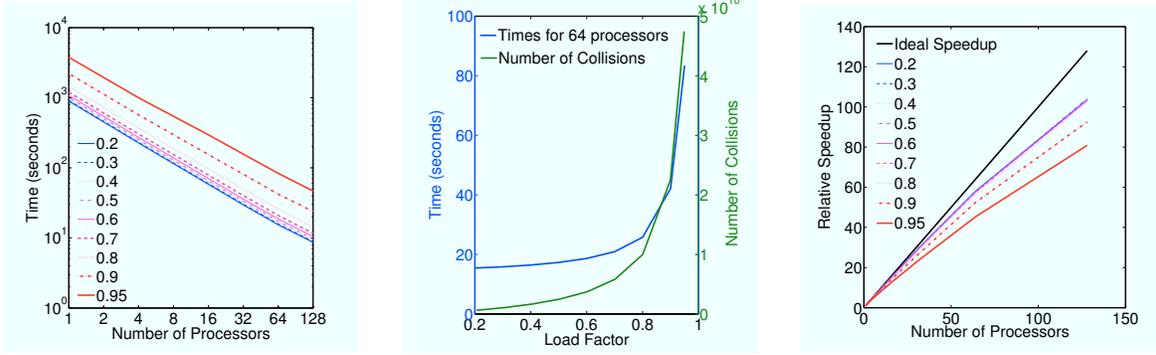
The bitwise $\&$ with $2^{53} - 1$ is necessary to avoid the performance hit arising from the current Cray XMT implementation that handles division of integers greater than 53 bits with a software routine rather than as a hardware instruction. Observe that the large, prime constant has the effect of separating the hash function values of adjacent integer keys.

Figure 7(a) shows the time to insert the five billion integers. As expected, insertion time is inversely proportional to l . For $0.2 < l < 0.7$, performance falls within a tight band, but afterwards quickly degrades for $l > 0.7$. Figure 7(b) keeps the number of processors constant at 64, and shows plots for both insertion time and number of collisions. As expected, it shows a strong relationship between number of collisions and insertion time. In terms of speedup, shown in Figure 7(c), the curves are nearly identical for $l \in [0.2, 0.7]$, but degrades for $l \geq 0.8$. Between $l = 0.2$ and $l = 0.95$ there is a 21.9% drop in speedup for 128 processors.

We see in Figure 8 that for light loads, linear probing performs slightly better than HACHAR, especially as the number of processors grow past 64. However, for lower processor counts, HACHAR handles large load factors significantly better than Linear Probing.

B. Integers, Zipfian Distribution

To test performance on a power law distributed set of integers, we generated a file consisting of integers following Zipf’s law arranged by a random shuffling. Zipf’s law, named after George Kingsley Zipf [15], is the empirical observation that, given some corpus of natural language utterances, the frequency of a word is inversely proportional to its rank. Thus, the frequency of a word with rank k can be approximated by $f(k, s, N) = \frac{1}{k^s H_{N,s}}$ where N is the number of distinct words in a corpus, s is an exponent, usually close to unity, and $H_{N,s}$ is the N th generalized harmonic number. If we set $s = 1$, then we arrive at the following equation for



(a) This figure shows the time in seconds for inserting five billion integers as a function of time and load factor.

(b) This figure compares the number of collisions with time needed to insert the data into the hash set. Performance appears tightly coupled with the number of collisions. Pairing the number of collisions with the reported times results in a correlation coefficient of 0.997.

(c) This figure presents the relative speedup obtained by varying the number of processors and load factor. The speedup curve is largely identical for load factors up to 0.7, and degrades thereafter. Between $l = 0.2$ and $l = 0.95$, there is a 21.9% drop for the 128 processor case.

Fig. 7: Linear Probing Results on five billion uniformly distributed random integers.

the count of the rank k item:

$$c(k) = \left\lfloor \frac{c(1)}{k} \right\rfloor \quad (3)$$

where the flooring function is necessary to insure an integer value. We use Equation 3 to construct a set of integers that follows Zipf’s law by setting $c(1)$ to be 250 million, resulting in $S_{\text{Zipfs}} \approx 4.969 \times 10^9$ and $U_{\text{Zipfs}} = 250 \times 10^6$. We also randomly shuffled the data.

We thought that a slightly different hash function that relies upon bitwise $\&$ rather than a division or modulus operation would be more efficient. So we tried a slightly different hash function for the Zipfian integers than the function used for the uniform random data set. Instead of allowing arbitrary array sizes, we confine $S_{\text{array}} \in \{2^x | x \in \mathbb{Z}^+\}$, which allows us to create a bit mask, $S_{\text{array}} - 1$ that we can use to obtain the index into the array. Thus we still employ Equation 1, but instead of using Equation 2, we use the following:

$$\text{getArrayIndex}(x) = \text{hash}(x) \& (S_{\text{array}} - 1) \quad (4)$$

In a limited number of trials, using $S_{\text{array}} - 1$ to determine the array location from the hash performs slightly better ($\sim 1\%$) than the modulus operator. It may be that this slight difference in time measurements may be due to experimental error.

1) *Results:* Using Equation 4 results in no collisions when $S_{\text{array}} \geq U_{\text{Zipfs}}$. Thus, for the linear probing method, this experiment is primarily a test of how well the method handles power law data. For HACHAR, we allow $S_{\text{array}} < U_{\text{Zipfs}}$ to force collisions. Figure 9 compares Linear Probing with $S_{\text{array}} = 2^{28}$, and HACHAR with $S_{\text{array}} \in \{2^x | x \in \{25, 26, 27, 28\}\}$. When collisions are absent, i.e. $S_{\text{array}} = 2^{28}$ and $l \approx 0.93$, the results for both strategies are almost identical, except for the case when the number of processors is

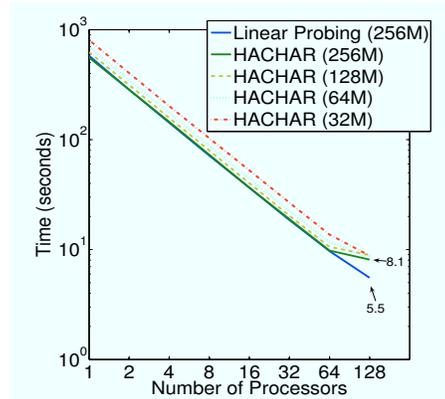


Fig. 9: Performance comparison of the two strategies on the Zipf integer data set. Performance is nearly identical for both linear probing and HACHAR, though HACHAR loses scalability with 128 processors.

equal to 128. The HACHAR method loses scalability at that point. Also, as we shrink S_{array} , we force collisions and performance degrades for HACHAR, but gracefully. Regardless of the load factor, HACHAR retains scalability up through 64 processors, though all appear to hit a bottleneck at 128 processors.

C. Wikipedia

Our final experiment involves finding the global word count within a snapshot of Wikipedia. The snapshot we used has $S_{\text{Wikipedia}} \approx 1.42 \times 10^9$ and $U_{\text{Wikipedia}} \approx 16.3 \times 10^6$. Like most text corpora, this instance approximates Zipf’s law, and presents a good test to gage how well we handle power law data.

1) *Results:* Figure 10 presents a comparison between linear probing and HACHAR on the Wikipedia instance. The time reported is for insertion only; it does not include time for tokenizing the data (splitting the input stream into strings that were separated by whitespace).

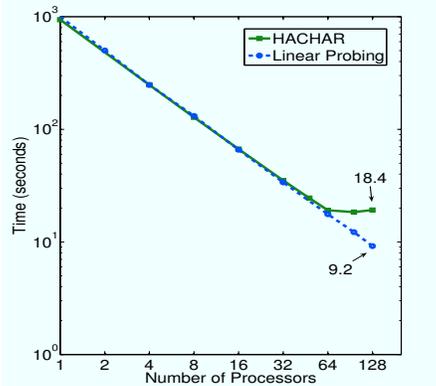


Fig. 10: The figure compares performance of the two strategies on a Wikipedia instance. Similar to other experiments, performance is nearly identical for both linear probing and HACHAR, though again HACHAR suffers scalability problems after 64 processors.

The hash table size for linear probing is 64M slots and for HACHAR is 32M. Similar to previous experiments, HACHAR and Linear Probing perform nearly identical up to 64 processors. After 64, the scalability of HACHAR degrades.

IV. CONCLUSIONS AND FUTURE WORK

We have adapted two standard hashing strategies — linear probing and hashing with chaining— for efficient execution on the Cray XMT. The experiments show that our adaptations allow for good scaling. For those applications where the number of keys is bounded, linear probing works well since the hash table can be made very large on an XMT and the two-step location acquiring strategy we described does minimize memory contention. For applications without a known upper bound, our HACHAR method works reasonably well. Also note that HACHAR is somewhat more complex to implement than linear probing.

Tracing the HACHAR code on 128 processors show the only long latency traps coming from line 2 in procedure `InsertIntoEmptyBucket` (Figure 5), which is completely bypassed once the bucket contains at least one key. Thus, the nominal case is not on the path containing the hotspot, meaning the initial populating of buckets is the root cause of the scaling limitation.

We would like to compare these two strategies against a “thread-local (contention-free) followed by an aggregation” strategy (the name *hash-reduce* may be appropriate for this strategy). We expect that a hash-reduce strategy will have better scale further than the strategies explored in this paper.

It would be interesting to investigate the impact that supporting a “remove” operation would have on the scalability of both of our hashing strategies.

ACKNOWLEDGMENTS

The authors would like to thank Jonathan Berry for suggesting this collaboration and to thank Bob Adolf for his thorough review of our drafts.

This work was funded under the Center for Adaptive Supercomputing Software - Multithreaded Architectures (CASS-MT) at the Dept. of Energys Pacific Northwest National Laboratory. Pacific Northwest National Laboratory is operated by Battelle Memorial Institute under Contract DE-ACO6-76RL01830.

REFERENCES

- [1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, “Introduction to algorithms, second edition,” 2001.
- [2] D. Knuth, *The Art of Computer Programming: Sorting and Searching*, 2nd ed. Addison-Wesley, 1998, vol. 3.
- [3] S. H. Bokhari and J. R. Sauer, “A parallel graph decomposition algorithm for DNA sequencing with nanopores,” *Bioinformatics*, vol. 21, no. 7, pp. 889–896, 2005. [Online]. Available: <http://bioinformatics.oxfordjournals.org/cgi/content/abstract/21/7/889>
- [4] J. Cieslewicz, J. Berry, B. Hendrickson, and K. A. Ross, “Realizing parallelism in database operations: insights from a massively multithreaded architecture,” in *DaMoN '06: Proceedings of the 2nd international workshop on Data management on new hardware*. New York, NY, USA: ACM, 2006, pp. 4+. [Online]. Available: <http://dx.doi.org/10.1145/1140402.1140408>
- [5] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny, “Software and algorithms for graph queries on multithreaded architectures,” *Parallel and Distributed Processing Symposium, International*, vol. 0, p. 495, 2007.
- [6] A. R. Karlin and E. Upfal, “Parallel hashing—an efficient implementation of shared memory,” in *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*. New York, NY, USA: ACM, 1986, pp. 160–168.
- [7] H. Lee, “Parallel hashing algorithms on BSP and QSM models,” in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, April 2004, pp. 175–.
- [8] P. Larson and M. Krishnan, “Memory allocation for long-running server applications,” in *Proceedings of the 1st international symposium on Memory management*. ACM New York, NY, USA, 1998, pp. 176–185.
- [9] M. Tofte and J.-P. Talpin, “Region-based memory management,” *Inf. Comput.*, vol. 132, no. 2, pp. 109–176, 1997.
- [10] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney, “Region-based memory management in cyclone,” in *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. New York, NY, USA: ACM, 2002, pp. 282–293.
- [11] M. Tofte, L. Birkedal, M. Elsmann, and N. Hallenberg, “A retrospective on region-based memory management,” *Higher Order Symbol. Comput.*, vol. 17, no. 3, pp. 245–265, 2004.
- [12] J. Feo, D. Harper, S. Kahan, and P. Konecny, “ELDORADO,” in *CF '05: Proceedings of the 2nd conference on Computing frontiers*. New York, NY, USA: ACM, 2005, pp. 28–34.
- [13] D. Chavarría-Miranda, A. Marquez, J. Nieplocha, K. Maschhoff, and C. Scherrer, “Early Experience with Out-of-Core Applications on the Cray XMT,” in *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium*, April 2008, pp. 1–8.
- [14] S. Kahan and P. Konecny, ““MAMA!”: a memory allocator for multithreaded architectures,” in *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2006, pp. 178–186.
- [15] G. K. Zipf, *Psycho-Biology of Languages*. Houghton-Mifflin, 1935.