ABSTRACT

Hashing is a fundamental technique in computer science to allow $O(1)$ insert and lookups of items in an associative array. Here we present several novel thread coordination and hashing strategies and compare and contrast their performance on large, shared memory symmetric multiprocessor machines, each possessing between a half to a full terabyte of memory. We show how our approach can be used as a key kernel for fundamental paradigms such as dynamic programming and MapReduce. We further show that a set of approaches yields close to linear speedup for both uniform random and more difficult power law distributions. This scalable performance is in spite of the fact that our set of approaches is not completely lock-free. Our experimental results utilize and compare an SGI Altix UV with 4 Xeon processors (32 cores) and a Cray XMT with 128 processors. On the scale of data we addressed, on the order of 5 billion integers, we show that the Altix UV far exceeds the performance of the Cray XMT for power law distributions. However, the Cray XMT exhibits greater scalability.

Keywords
Hashing, MapReduce, NUMA, SMP, Cray XMT, SGI Altix UV

1. INTRODUCTION

In this paper, we present a set of novel thread coordination and hashing strategies. We evaluate their performance empirically on two large, shared memory systems: an SGI Altix UV with 4 Xeon processors (32 cores) and a half terabyte of main memory, and a Cray XMT with 128 Threadstorm processors and one terabyte of memory. The particular use case we investigate is that of batch insertion of a large set of key-value pairs, $(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)$, into a hash table, where each insertion results in an update to the currently stored value. For example, when inserting the pair, $(k_i, v_i)$, the process is as follows:

1. Calculate the hash of the key according to the selected hash function: $h_{k_i} = \text{hash}(k_i)$. For simplicity, we will ignore, for the moment, the possibility of hash collisions.

2. At location $h_{k_i}$, there is an associated value, $\text{table}[h_{k_i}].value$. We then perform an update operation, $u_i$, that must be both associative and commutative, $\text{table}[h_{k_i}].value \leftarrow u(\text{table}[h_{k_i}].value, v_i)$

We call this operation of updating a key’s stored value with additional information as an updateInsert with three parameters, a key, a value, and the associative and commutative update function. Note that, while we focus on the updateInsert operation for this paper, the framework we present easily extends to a vanilla insert operation (i.e. if a value already exists for a given key, the insert results in a no-op). Also, lookups proceed similarly to the above procedure except, instead of updating the value, we simply return it. We do not handle the case of deletions. We have investigated extensions that allow for deletions, but they add overhead and are not necessary for the use cases we address in this paper.

The tasks we target are in support of two fundamental programming paradigms: dynamic programming [1] and MapReduce [4].

The basic idea behind dynamic programming is to compute the optimal solution to a complex problem by 1) breaking it down into subproblems, 2) finding the optimal solution to each subproblem, and 3) combining them. Often, many of these subproblems are identical; one of the speedups obtained by dynamic programming is through avoiding duplicated computation. One way to do this is with a hash table that supports inserts and lookups [16]. The idea is to perform a “top-down” approach with recursion, starting from the original problem and diving down into subproblems. The solutions of subproblems are stored in a hash table. Thus, threads can first check the hash table to see if a subproblem has already been solved. If so, it can simply use the answer stored in the hash table. Otherwise, it will perform the work and insert the solution in the hash table for consumption by others.

To see how batch inserts (and lookups) can be applied in a dynamic programming setting, first consider the parallelization scheme by Stivala et al. [16] presented below in Figure 2. The pseudo-code follows this recursive scheme:

$$f(x) = \begin{cases} g(x) & \text{if } b(x) \text{ then } \text{ else } F(f(x_1), \ldots, f(x_n)) \end{cases}$$

The function $b(x)$ returns true if a base case is satisfied,
MapReduce from the backdrop of the originally targeted architecture, that of distributed memory clusters. We will then discuss how these concepts map to a shared memory design.

Figure 1 shows the overall process. Raw data is partitioned and given to a set of processes each executing the same function, named the \textit{map} function. Each \textit{map} process scans through its partition of data and emits key-value pairs \((k,v)\). A common example application for MapReduce is to count word frequencies within a document corpus. The \textit{map} function, in this case, scans through a portion of the document corpus. Any time a word is encountered, the \textit{map} function emits the word as the key and sets the value to be one, indicating one occurrence of the word.

We will defer discussion of the \textit{combine} phase for a moment and talk about the \textit{reduce} phase. All of the key-value pairs emitted during \textit{map} are collated together, so that all of the values associated with a given key are co-located on a single node. The \textit{reduce} function is then applied to the set of values for a given key. For the example of counting words, the \textit{reduce} function is a simple summation of the value set for a given key.

This collation of the key-value pairs, sorting on the keys, can be very expensive in terms of network bandwidth if one were to simply broadcast all key-value pairs. Also, for power law distributions, the number of pairs for commonly occurring keys may exceed the working memory of a single node. For instance, the word the dominates the distribution of words for English language corpora, and sending all the instances of the word the to one single node is not advisable.

A common way to address this is with a \textit{combine} function. The \textit{combine} function, which is usually identical to the \textit{reduce} function, is applied locally at each node to the set of key-value pairs emitted by the local \textit{map} process. Thus, instead of each node sending a large number of key-value pairs over the network for the word the, using the \textit{combine} function allows each node to emit only one pair, the total summation of the local to the node. This only works if the \textit{reduce} operation is both commutative and associative. Otherwise, all pairs must be sent and collected together by key in their entirety. The restriction of commutativity and associativity for the \textit{reduce} function still lends itself to a wide range of applications, and this assumption is employed in the design of higher-level languages built on top of MapReduce, such as Sawzall [13].

By considering only \textit{reduce} operations that are both commutative and associative, on a shared memory machine we can completely avoid the collation of key-value pairs. Also, we can execute the \textit{reduce} function whenever the \textit{map} func-
tion emits a key-value pair, in essence performing reduction in place, collapsing the map and reduce phases together. This is accomplished by means of a global hash table. For each key-value pair emitted, we call `updateInsert(key, value, update)`, where `update` is the reduce operation that we perform on the currently stored value in the hash table and the `value`. Thus, to port the concept of MapReduce onto a shared memory architecture for commutative and associative reduce operations, all you need is a hash table. Many algorithms are being developed for MapReduce on clusters. The hash table implementation we present here can be used as a kernel for porting these algorithms quickly to shared memory platforms.

Another important factor we address in this paper is the underlying distribution of the data, ensuring that, regardless of the distribution, performance and scalability are similar. Uniform random distributions are most often addressed in the literature [15], [12], [14], [10]. However, many important, real-world problems involve power law distributions. To name but a few: natural language corpora [18], the occurrence of substrings in DNA sequences [11], and the distribution of degrees in internet graphs [3]. We present a strategy that scales well on both uniform random and power law distributions.

2. PREVIOUS WORK

In the past decade, there has been significant interest in developing lock-free algorithms and data structures. The term lock-free encapsulates the notion that no thread can block system-wide progress. Thus, if any thread is delayed, it does not delay or impact the progress of other threads. A significant area of focus is that of transactional memory [9], which allows groups of memory operations to operate atomically, thus simplifying the programming paradigm.

Besides transactional memory, much research in lock-free algorithms has focused on atomic compare-and-swap operation, CAS, provided by many architectures. The function, taking three parameters, proceeds as follows. If the value in the memory location specified in the first parameter is equal to the value of second parameter, then the memory location is updated with the value of the third parameter. Otherwise, the memory location is left unmodified. Regardless of whether an update occurred, the content before the CAS operation is returned.

Gao et al. [7] present a lock-free synchronization primitive using CAS that avoids theABA problem. The ABA problem is one where a thread reads a value A, and then performs CAS on the location of A to set it to another value. The intent is to only execute the change if A did not change between the read and the CAS call. However, between the read and CAS call, another thread may have changed A to B and then back to A, thus breaking the intended logic. In another paper, Gao et al. [6] present a robust, lock-free, open addressing scheme for hash tables that can operate effectively in heterogeneous environments. Shalev and Shavit [15], using CAS once again, develop an extensible hash table. Fraser and Harris [5] extend CAS to implement a multiword compare-and-swap that can operate on an arbitrary set of memory locations simultaneously, and also present other functions to aid in development of lock-free data structures. One guiding principle they employ is that of disjoint-access parallelism, which means that updates to non-overlapping locations can be performed concurrently. We use this principle in all of our synchronization primitives, as will be explained next.

Our work presents four synchronization primitives, only one of which is completely lock-free. However, we find this to not be much of an issue, as performance is dictated by other factors. To understand this, consider that the hash table can be thought of as a large array, where each item, or bucket, in the array is used to store the data for one key. We perform fine-grained locking at the level of the bucket. Thus, if one thread locks out a bucket, it only affects other threads trying to access that bucket. Furthermore, we only perform locking when a bucket is first declared occupied for a particular key. Access to the bucket is otherwise lock-free once the bucket’s key has been set.

3. SUPERCOMPUTERS UNDER STUDY

This section describes the two machines we use for this study: the Cray XMT and the SGI Altix UV.

3.1 Cray XMT

The Cray XMT is a unique shared memory machine with multithreaded processors especially designed to support fine-grained parallelism and perform well despite memory and network latency. Each of the custom-designed compute processors (called Threadstorm processors) comes equipped with 128 hardware threads, called streams in XMT parlance. The processor, instead of the operating system, has responsibility for scheduling the streams. To allow for single-cycle context switching, each stream has a program counter, a status word, eight target registers, and 32 general-purpose registers. At each instruction cycle, an instruction issued by one stream is moved into the execution pipeline. The large number of streams allows each processor to avoid stalls due to memory requests to a much larger extent than commodity microprocessors. For example, after a processor has processed an instruction for one stream, it can cycle through the other streams before returning to the original one, by which time some requests to memory may have completed. Each Threadstorm can currently support 8 GB of memory, all of which is globally accessible. The system we use in this study has 128 processors and 1 TB of shared memory. The processor speed of the Threadstorm processors is 500 MHz.

Programming on the XMT consists of writing C/C++ code augmented with non-standard language features including generics, intrinsics, futures, and performance-tuning compiler directives such as pragmas. Generics are a set of functions the Cray XMT compiler supports that operate atomically on scalar values, performing either `read`, `write`, `purge`, `touch`, or `int_fetch_add` operations. Each 8-byte word of memory is associated with a full-empty bit (FEB) and the read and write operations interact with these bits to provide light-weight synchronization between threads. We refer to the set of operations interacting with the FEB as FEB semantics. Here are some examples of the generics provided:

- **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. The stream is blocked if the bit is in the empty state.
• **readFF**: Returns the value of a variable when the variable is in a full state, and leaves the bit as full.

• **writeFF**: Writes a value to a variable if the variable is in the empty state, and simultaneously sets the bit to be full. The stream is blocked if the bit is in the full state.

• **int_fetch_add**: Atomically adds an integer value to a variable.

Parallelism is achieved explicitly through the use of futures, or implicitly when the compiler attempts to automatically parallelize for loops. Futures allow programmers to explicitly launch threads to perform some function. Besides explicit parallelism through futures, the compiler attempts to automatically parallelize for loops, enabling implicit parallelism. The programmer can also provide pragmas that give hints to the compiler on how to schedule iterations of the for loop on various threads, which can be by blocks, interleaved, or dynamic. In addition, the programmer can supply hints on how many streams to use per processor, etc. We extensively use the \#pragma mta for all streams i of n construct that allows programmers to be cognizant of the total number of streams that the runtime has assigned to the loop, as well as provide an iteration index that can be treated as the id of the stream assigned to each iteration.

The XMT attempts to make memory access look uniform through its hyper-threaded architecture. However, there are two aspects that a programmer can utilize to exploit the NUMA aspects of the machine. The first is that accessing memory locations local to the processor is faster than accessing from non-local. The second is that each processor has a memory controller with a 128KB buffer. If one can keep the content of that buffer relatively stable and unchanging, then one can perform memory operations at the clock rate of the processor.

Many of the results outlined later utilize code from the MultiThreaded Graph Library (MTGL)\(^1\) [2], an open source library with data structures and algorithms specifically designed to run scalably on shared-memory platforms such as the XMT.

### 3.2 SGI Altix UV

The SGI Altix UV is another large-scale, shared memory system but, unlike the XMT, uses industry standards and commodity parts such as Xeon processors. It is a cache-coherent system. Since industry standards are supported, one can use standard C/C++ instead of the more specific generics and intrinsics of the XMT. Also, the operating system is Linux. To get parallelism, we used a combination of OpenMP\(^2\) and Qthreads\(^3\) [17]. The latter is an open source library that presents a general API for thread management and coordination that hides machine-specific implementation details. The Altix UV comes in three different flavors, the 10, the 100, and the 1000, the difference being to what degree the architecture can scale. The Altix UV 10 is limited to 4 Xeon processors and a terabyte of memory, while the 100 can have up to 256 processors and 16 terabytes. The machine we use is the Altix UV 10. It has 4 Xeon X7550 processors that run at 2.00 GHz and has 18 total megabytes of cache per processor. Each processor has 8 cores and 16 total threads. The amount of memory for the system is half a terabyte.

Some important considerations for this architecture and software stack are the memory hierarchy and the paging system. The Altix UV is much more susceptible to memory latency issues than the XMT. To get scaling, one must design the algorithm to exploit the caches, and keep memory accesses as close as dedicated to each processor as possible. Another factor that comes up in our experiments is the memory paging system used by Linux. Linux is a demand-based system, meaning that pages of memory are not mapped into physical memory until first referenced. We found that the process of mapping these pages into memory does not scale well, and that a process for priming memory is needed in order to get good scaling on algorithm run time.

### 4. Thread Coordination

This section discusses the mechanisms we use for thread coordination to ensure algorithmic correctness. We discuss these in terms of two different situations, that of known and unknown key distributions. For all of the hashing strategies we employ, we use open addressing and linear probing. Basically the hash table can be thought of as a large array. Hashing provides an index into the hash table array, and the process of inserting or updating the value for a key probes from the original index until it finds the appropriate location, which is described in more detail below.

#### 4.1 Known Key Distributions

We first deal with the case where the key distribution, \(K\), is known and, in particular, where we know that a certain key, \(k_u\), will never be inserted into the hash table, i.e. \(k_u \notin K\). This is useful for cases when ids are assigned to a data set, and the curator of the data has control over how ids are assigned to particular data items. In such situations we can use \(k_u\) to signify that a bucket in the hash table is unclaimed.

As explained previously, we use open addressing on a large array, \(table\), for our hash table data structure. For this case, each bucket in the array is a pair, \((key, value)\). To initialize the table, for each \(i\) in \([0, |table| - 1]\), we set \(table[i].key = k_u\). Also, since we are performing a counting exercise for our experiments, we set \(table[i].value = 0\).

We will later discuss various strategies for organizing the data in order to exploit the NUMA aspects of the architectures under study, but at the heart of inserting the data is the procedure in Algorithm 1.

Line 2 performs the hash function:

\[
hash(x) = (x \cdot C) \mod |table| \tag{1}
\]

where \(C = 31, 280, 644, 937, 747\) is a large prime constant (taken from \([8]\)). There are many candidates for hash functions, but that is not the focus of this work. This simple hash function was sufficient for our needs. Line 3 records the results of the hash into another variable, \(start\), the index from which we linearly probe forward until the correct bucket is found. Should the end of the array be reached, we wrap around to the beginning. If the table is ever completely filled, the repeat loop exits. However, for the use-case under which we are operating, the size of the data to be inserted is known beforehand. Managing the load factor of the table is relatively straightforward and this degenerate case will never be reached.
In the body of the repeat loop, the first step on line 5 is to call \texttt{CAS} on the \textit{j}th key location in the hash table array. On
line 6, we examine the results of the \texttt{CAS} operation. If the
previous value, \texttt{prev}, is \textit{k}_u, then we know that this is the first
thread to ever touch this bucket of the hash table array. In
essence, the thread claimed the bucket for the thread’s cur-
rent key, \textit{k}. It can then perform an atomic update. For our
application, that of counting keys, we increment the value
portion of the table array by 1. This addition operation has
an atomic variant for both the XMT and the UV. If an op-
eration is desired that cannot be updated atomically, then a
fine-grained locking mechanism would need to be employed
to update \texttt{table[j].value}.

Also on line 6, if \texttt{prev} is not \textit{k}_u, then this thread is not the
first to touch the \textit{j}th element of the table array. However,
if the assigned key in the \textit{j}th is the same as the key under
consideration, the thread can still proceed with the atomic
update to the value.

Note that, since we make use of the atomic \texttt{CAS} operation,
this approach is lock-free, assuming that the update proce-
dure is a non-blocking operation or a set of non-blocking
operations.

The Cray XMT lacks an atomic \texttt{CAS}, thus we present an
implementation specific for its needs, using FEB semantics,
in Algorithm 2. Line 5 first checks to see if the bucket has
an associated key. If so, it then checks to see if the bucket’s
key matches the current key. If so, it can then perform the
update. Otherwise, the thread continues to probe for-
ward. Note that this is a non-blocking read of the bucket’s
key. After a bucket is declared as occupied, access to the
bucket is non-blocking, assuming the update operation is
non-blocking.

If the comparison on line 5 turns out to be false, indi-
cating that a bucket does not have an assigned key, the
thread attempts to claim the bucket, on line 11, with a call to
\texttt{readFE} on the bucket’s key. The first thread to gain con-
tral of bucket’s key will set it to the current key on line
13. All other threads will have to check if the key assigned
while they were waiting matches their current key. If so,
they perform the update, otherwise they continue to probe.

Note that while this is a blocking algorithm, it only blocks
a specific bucket while the bucket is being claimed for a key.
It should be noted that the key type is limited to data types
that are 8-bytes or larger, else the full/empty bits do not align properly.

Algorithm 1 Update Insert Process with Known Key Dis-
tributions using \texttt{CAS}

1: \textbf{procedure} \texttt{updateInsert}(\texttt{table, k, v, atomicUpdate})
2: \hspace{1em} \texttt{j} \leftarrow \texttt{hash}(\texttt{k})
3: \hspace{1em} \texttt{start} \leftarrow \texttt{j}
4: \hspace{1em} \texttt{repeat}
5: \hspace{2em} \texttt{prev} \leftarrow \texttt{cas} (\texttt{table [j].key, k}, \texttt{k}_u, \texttt{k})
6: \hspace{2em} \textbf{if} \texttt{prev} \texttt{==} \texttt{k}_u \texttt{||} \texttt{table [j].key} \texttt{==} \texttt{k} \textbf{then}
7: \hspace{3em} \texttt{atomicUpdate} (\texttt{table [j].value, 1})
8: \hspace{2em} \texttt{break}
9: \hspace{2em} \texttt{end if}
10: \hspace{2em} \texttt{j} \leftarrow \texttt{j} + 1
11: \hspace{2em} \textbf{if} \texttt{j} \texttt{==} \texttt{|table|} \textbf{then}
12: \hspace{3em} \texttt{j} \leftarrow 0
13: \hspace{2em} \texttt{end if}
14: \hspace{2em} \texttt{end if}
15: \hspace{1em} \texttt{until} \texttt{j} \texttt{==} \texttt{start}
16: \textbf{end procedure}

Algorithm 2 Update Insert Process with Known Key Dis-
tributions using FEB semantics

1: \textbf{procedure} \texttt{updateInsert}(\texttt{table, k, v, atomicUpdate})
2: \hspace{1em} \texttt{j} \leftarrow \texttt{hash}(\texttt{k})
3: \hspace{1em} \texttt{start} \leftarrow \texttt{j}
4: \hspace{1em} \texttt{repeat}
5: \hspace{2em} \textbf{if} \texttt{table [j].key} \texttt{!=} \texttt{k}_u \texttt{then}
6: \hspace{2em} \textbf{if} \texttt{table [j].key} \texttt{==} \texttt{k} \textbf{then}
7: \hspace{3em} \texttt{atomicUpdate} (\texttt{table [j].value, v})
8: \hspace{2em} \texttt{break}
9: \hspace{2em} \texttt{end if}
10: \hspace{2em} \texttt{else}
11: \hspace{3em} \texttt{probe} \leftarrow \texttt{readFE} (\texttt{table [j].key})
12: \hspace{3em} \textbf{if} \texttt{probe} \texttt{==} \texttt{k}_u \texttt{then}
13: \hspace{4em} \texttt{writeFE} (\texttt{table [j].key, k}, \texttt{v})
14: \hspace{4em} \texttt{atomicUpdate} (\texttt{table [j].value, v})
15: \hspace{3em} \texttt{break}
16: \hspace{3em} \texttt{else}
17: \hspace{4em} \texttt{writeFE} (\texttt{table [j].key, k}, \texttt{probe})
18: \hspace{4em} \textbf{if} \texttt{probe} \texttt{==} \texttt{k} \texttt{then}
19: \hspace{5em} \texttt{atomicUpdate} (\texttt{table [j].value, v})
20: \hspace{4em} \texttt{break}
21: \hspace{3em} \texttt{end if}
22: \hspace{3em} \texttt{end if}
23: \hspace{2em} \texttt{end if}
24: \hspace{2em} \texttt{j} \leftarrow \texttt{j} + 1
25: \hspace{2em} \textbf{if} \texttt{j} \texttt{==} \texttt{|table|} \textbf{then}
26: \hspace{3em} \texttt{j} \leftarrow 0
27: \hspace{2em} \texttt{end if}
28: \hspace{1em} \texttt{until} \texttt{j} \texttt{==} \texttt{start}
29: \textbf{end procedure}

4.2 Unknown Key Distributions

For the more general case of where the key distribution is
unknown and we cannot specify with certainty that a partic-
ular \textit{k} is not in \textit{K}, another strategy is warranted. Now, each
bucket in the hash table array is a triple, (\textit{k}, \textit{value}, \textit{lock}). The new variable, \texttt{lock}, is generally either a 32- or
64-bit integer, depending on the smallest size that can be oper-
oned on with either \texttt{CAS} or FEB semantics. It handles thread
synchronization when a bucket is declared claimed. As be-
fore, the table is initialized so that each \textit{i} \in \{0, |table| - 1\},
\texttt{table[i].value} \leftarrow 0. However, for this case, no special con-
sideration is needed for the \textit{key} field. The \texttt{lock} variable
now indicates the status of a bucket, and the \textit{key} field is
set based upon the status of \texttt{lock}. For initialization, each
\texttt{table[i].lock} \leftarrow \texttt{UNCL} where \texttt{UNCL} indicates the bucket is un-
claimed. The other two possible values are \texttt{CL} and \texttt{INTER},
signifying claimed and intermediate, respectively. The up-
date\texttt{Insert} process when atomic \texttt{CAS} and an atomic update
are available is outlined in Algorithm 3.

As we discuss the outline of this procedure, note that the
general outline is the same as that of Algorithm 2. The main
difference being that we use \texttt{CAS} instead of FEB semantics to
perform blocking to ensure that only one thread can claim a
bucket.

The first conditional within the body of the repeat loop
checks to see if a bucket is claimed and, if so, checks to make
sure the key to be updated matches the key of the bucket.
### Algorithm 3 Update Insert Process with Unknown Key Distributions using CAS

1. **procedure** `updateInsert(table, k, v, atomicUpdate)`
2. \( j \leftarrow \text{hash}(k) \)
3. \( \text{start} \leftarrow j \)
4. repeat
5. \[ \text{if } \text{table}[j].\text{lock} == \text{CL} \text{ then} \]
6. \[ \text{if } \text{table}[j].\text{key} == k \text{ then} \]
7. \( \text{atomicUpdate(table}[j].\text{value}, v) \)
8. \( \text{break} \)
9. \( \text{end if} \)
10. \( \text{else} \)
11. repeat
12. \( \text{prev} \leftarrow \text{CAS(table}[j].\text{lock}, \text{UNCL}, \text{INTER}) \)
13. until \( \text{table}[j].\text{lock} != \text{CL} \wedge (\text{prev} != \text{UNCL} \vee \text{table}[j].\text{lock} == \text{INTER}) \)
14. \( \text{if } \text{table}[j].\text{lock} == \text{INTER} \text{ then} \)
15. \( \text{if } \text{table}[j].\text{lock} == \text{CL} \text{ then} \)
16. \( \text{table}[j].\text{key} \leftarrow k \)
17. \( \text{atomicUpdate(table}[j].\text{value}, v) \)
18. \( \text{break} \)
19. \( \text{else} \)
20. \( \text{if } \text{table}[j].\text{key} == k \text{ then} \)
21. \( \text{atomicUpdate(table}[j].\text{value}, v) \)
22. \( \text{break} \)
23. \( \text{end if} \)
24. \( \text{end if} \)
25. \( j \leftarrow j + 1 \)
26. until \( j == |\text{table}| \)
27. \( j \leftarrow 0 \)
28. \( \text{end if} \)
29. until \( j == \text{start} \)
30. end procedure

In that case, the thread can perform the update to the value associated with the bucket. If the keys do not match, the thread continues probing forward. If the bucket is unclaimed, the thread attempts to claim it for key \( k \). The thread enters a blocking loop on line 12 that attempts to use CAS to change the lock variable from unclaimed to intermediate. The thread exits this loop when either it successfully sets lock to intermediate or lock has been set to claimed by another thread. If the thread successfully set lock to intermediate (note that only one thread can succeed in this regard), it then sets the bucket’s key variable to \( k \) and sets lock to be claimed. Also, it performs the atomic update to value. Note that, since key is updated before lock is set to claimed, we are guaranteed that any threads entering the first conditional on line 5 will have the necessary information. In other words, if a thread finds a bucket claimed, then its key has also been assigned. After exiting the repeat loop of line 11, if a thread finds that the lock variable is not in the intermediate state but is already claimed, it then proceeds to check key to see if it matches \( k \). If so, the value is updated. Otherwise, the thread continues probing. The algorithm using FEB semantics is presented in Algorithm 4. This particular strategy was presented earlier in [8], but we outline here for the sake of completeness. The procedure has a similar flow to the one using CAS. This time there is no explicit repeat loop that handles blocking the threads when a bucket is in the process of being claimed. For the XMT, this is handled at the hardware level.

### 4.3 Other Primitives

Besides `updateInsert`, there are two other functions that we use. The first one we will discuss is claiming a key for a bucket in a thread-dedicated table (Algorithm 5). The value associated with the bucket is untouched. Because the table is only accessed by one thread, we can ignore the thread synchronization concerns. In the interest of space, we only list the algorithm for known key distributions. For unknown key distributions, the procedure is similar except that the comparisons are done against the lock variable in the table array using the CL and UNCL constants. Also, the first conditional sets the lock variable to CL.

The `updateInsert` function inserts a key into the table if it does not exist. We have the need in one of our approaches for just the update procedure. If the key to be updated does not exist, the table is unchanged. Again, we are working in the context of a thread-dedicated table, so thread synchronization is not a concern. Algorithm 6 only presents the algorithm used when the key distribution is known. The function returns true when an update occurred, and false otherwise.
Algorithm 5 Claiming a Bucket with Known Key Distributions on a Thread-Dedicated Table

1: procedure LOCALCLAIMBUCKET(table, k)
2: $j ← \text{hash}(k)$
3: $\text{start} ← j$
4: repeat
5: if $\text{table}[j].key == k$ then
6: $\text{table}[j].key ← k$
7: break
8: else
9: if $\text{table}[j].key == k$ then
10: break
11: end if
12: end if
13: $j ← j + 1$
14: if $j == [\text{table}]$ then
15: $j ← 0$
16: else
17: until $j == \text{start}$
18: end procedure

Algorithm 6 Update-only with Known Key Distributions on a Thread-Dedicated Table

1: procedure LOCALUPDATE(table, k, v, atomicUpdate)
2: $j ← \text{hash}(k)$
3: $\text{start} ← j$
4: repeat
5: if $\text{table}[j].key == k$ then
6: return false
7: else
8: if $\text{table}[j].key == k$ then
9: $\text{update}(\text{table}[j], \text{value}, v)$
10: return true
11: end if
12: end if
13: $j ← j + 1$
14: if $j == [\text{table}]$ then
15: $j ← 0$
16: end if
17: until $j == \text{start}$
18: return false
19: end procedure

5. HASHING STRATEGIES

We now present two hashing strategies that employ the previously discussed insertion procedures as algorithmic primitives for inserting in batch a large set of key-value pairs into a hash table. The first strategy we discuss is a straightforward utilization of the primitives that ignores cache and NUMA-related concerns. We refer to this method as Cache-oblivious or CO for short. The second method tries to overcome the failings of CO by dealing first with frequently occurring keys in a thread-local manner before a global update occurs. We refer to this method as Removing heavy-hitters (RHH). We will refer to the combination of CO with the unknown key distribution strategy as CO_U and with the known key distribution as CO_K. Similarly for RHH_U and RHH_K.

5.1 Cache-oblivious

As stated earlier, the CO method is uncomplicated. Given a set of key-value pairs, KV, the method simply iterates through all of them, calling updateInsert. Our implementation uses block scheduling, meaning that each thread is assigned one contiguous block of key-value pairs. Since we use a simple open addressing scheme, the table array must be instantiated so that the load factor is beneath 0.7, otherwise performance degrades rapidly. We determine the correct size manually, but it would be relatively simple to sample the data first to determine an appropriate size.

Algorithm 7 Cache-oblivious

1: procedure BATCHINSERT(KV, update)
2: Create table array
3: for all $(k,v) ∈ KV$ do
4: updateinsert(table, k, v, update)
5: end for
6: end procedure

5.1.1 Performance

Here we discuss the performance of the CO strategy performing a counting exercise on integer data sets, i.e. the keys are integers, the value associated with each integer is 1, and the update function is addition. We examine four different data sets. The first data set is 5 billion uniform random integers grabbed from the range $[0, 2^{34} - 1]$. The last three are also 5 billion integer data sets, but drawn from a power law distribution. We use the Pareto distribution that has the following probability density function:

$$f(x) = \begin{cases} \frac{x^{-\alpha}}{x_{m}^{-\alpha+1}} & \text{for } x > x_m, \\ 0 & \text{for } x < x_m. \end{cases}$$

We let $x_m$ be 1 and we use three different values for $\alpha$: 0.25, 1, and 2. To get a set of integers, we discretized the values produced using the floor function. Because we found that the given hash function performed close to perfect when using the set of integers given by the initial distribution, we mapped each key to a random value in $[0, 2^{34} - 1]$. This was done to increase the likelihood of collisions in the hash table. For $\alpha = 0.25$, we arrived at a distribution that is close to Zipf’s law [18], i.e., that the frequency of the $n^{th}$ ranked item is twice that of the $2n^{th}$ ranked item. For $\alpha = 1, 2$, the curve becomes much steeper, with $\alpha = 2$ resulting in a difference of 6-8 between the $n^{th}$ and $2n^{th}$ ranked items.

For all of the results discussed in this paper, we present more detailed results at this page 4. The focus of this paper is on algorithm time; however, there are other important performance considerations such as file I/O. We did not attempt to optimize file I/O, but the times can be found on the site. In general, the XMT had a read I/O speed of about 300 MB/s and the UV approached 1 GB/s. On another XMT system with a richer I/O environment, we’ve seen speeds close to 4 GB/s. We feel that in either system, better I/O rates could have been obtained with further investment.

A more problematic overhead, at least for the UV, is that of data structure construction and priming. For the XMT, allocating memory for data structures presented a relatively

4https://software.sandia.gov/trac/MEGRAPHS-MC/wiki/SharedScalableHashing
small overhead cost on the order of a few seconds. However, the UV has a demand paging system that exacts a significant penalty the first time a location in memory is accessed. We found that to get good scaling during execution of the algorithm, we needed to first prime the data structures by touching each location in memory. This process of priming did not scale, and took on the order of 2-10 times more than the algorithm run time for higher thread counts. Some of these data initialization overheads could possibly be paid for only once by creating an allocation pool containing pre-initialized data structures from which processes could draw. We leave investigation of these concepts as future work.

Figure 3 displays the results of running CO_U and CO_K on both the XMT and the UV. First examining performance on the uniform random data set for the UV, it can be seen that the straight-forward approach of CO_U and CO_K scales reasonably well to 32 threads, with a relative speedup of 28.3 and an efficiency of 0.89. Going out to 64 threads, the max number supported by hardware, only increases the speedup to 33.3 with an associated efficiency of 0.52. This decrease in scalability as we expand to the limits of the hardware is to be expected. When 64 threads are used, the two threads per core share an execution unit, and scalability generally lessens or levels out for most applications.

CO fails completely on the UV for the power law distributed sets. Performance is best on a single thread and does not scale thereafter. This is likely due to cache thrashing as all the threads compete to write to the sections of the table that contain the frequent keys. For the XMT, the reader will notice that our results begin to drop off at 32 threads for the UV and at 64 threads for the XMT. This is likely due to cache thrashing as these data initialization overheads could possibly be paid for only once by creating an allocation pool containing pre-initialized data structures from which processes could draw.

We leave investigation of these concepts as future work.

5.2 Removing Heavy-Hitters

The RHH strategy avoids the weaknesses of the CO strategy by first removing frequently-occurring keys. The process is outlined by Algorithm 8 and is described in more detail below.

Besides a set of key-value pairs, KV, and an update function, the parameters of this algorithm include a fraction \( \alpha \in (0, 1] \) that determines how many key-value pairs are sampled from KV. Lines 2-5 perform necessary setup before the hashing begins. Line 3 creates an empty array of size \( n_{samples} \) for each thread that will store how many key-value pairs remain after sampling and thread-local processing.

At line 6, the algorithm enters a parallel region with five for loops. Lines 7-8 perform a manual calculation of the beginning and end point for the contiguous block assigned to each thread. Line 9 calculates the step size to get the proper number of samples.

The first for loop between lines 11-13 performs a sampling of the data. In this first pass, we only perform the local-ClaimBucket function on a local table, in essence keeping track of which keys are in the sample. The second for loop (15-23) then goes through and counts how many times each key occurs, but only if the key was sampled in the first loop. Any keys and their associated values that were not sampled are transferred into the KV2
array, but in a non-contiguous manner.

The third for loop (25-29) transfers all the key-value pairs from KV2 back into KV, but this time forming a contiguous block. These key-value pairs are all those that are not contained in the local table. This step isn’t strictly necessary. It is so we can repartition the remaining data and load-balance the work during the last loop.

The fourth loop (30-35) iterates through the content of the local table, and adds the entries to the global table. Loops 1-4 are what allow this method to scale well for power law distributions. All of the frequently occurring keys are largely addressed locally in a cache-friendly manner. When the update from the local table to the global table occurs, the number of operations for frequently occurring keys is on the order of the number of threads, rather than on the order of the count of the key, drastically reducing the amount of cache thrashing. At the end, the fifth loop (38-41) addresses any remaining keys that were not caught by the sampling procedure.

5.2.1 Performance

Figure 4 shows dramatic improvement for the UV under the RHH strategy. Now the UV is beating out the XMT for the power law distributed data on a thread to processor basis, and continuing to perform even with XMT on the uniform random data. For power law distributions, the UV is performing between 3.4 to 17.5 times better than the XMT on a thread to processor basis. Note that the performance is nearly identical with either a known or unknown key distribution. The relative scaling to 32 processors for the UV has a high of around 21-22 for $\alpha = 0.25$, but then decreases as $\alpha$ increases, coming to 10-11 for $\alpha = 2$. Another noticeable pattern is the increase in time as $\alpha$ becomes lower. The reason for this is that, as $\alpha$ becomes smaller, the number of keys increases, necessitating an increase in table size and a decrease in overall cache-friendliness. The UV’s performance increased dramatically for power law distributions,

Figure 4: The results of performing the Removing heavy-hitters strategy on sets of integers of size five billion for both the Altix UV and the Cray XMT. Figure (a) shows the results of performing with the unknown key distribution approach, and (b) shows the results with a known key distribution. The horizontal axes displays the number XMT processors used or the number total threads employed on the Altix UV. The XMT scales nearly linearly for all data sets, but the UV beats the XMT handily on power law distributions.

Algorithm 8 Removing heavy-hitters

```plaintext
Algorithm 8 Removing heavy-hitters
1: procedure batchInsert($KV, f, update$)
2: Create table array
3: Create KV2 array
4: nsamples ← $f\left(\frac{|KV|}{T}\right)$
5: nleft ← 0
6: for all threads $t$ of $T$ do
7: beg ← $(\frac{|KV|}{|T|}) * t_i$
8: end ← $(\frac{|KV|}{|T|}) * (t_i + 1) - 1$
9: step ← (end - beg)/nsamples
10: for $i ←$ beg; $i ≤$ end; $i ← i + step$ do
11: localClaimBucket ((local_table, KV[i].key)
12: end for
13: local_nleft ← 0
14: for $i ←$ beg, end do
15: result ← localUpdate(local_table, KV[i].key,
16: $KV[i]$).value, update)
17: if result == false then
18: KV2[local_nleft].key ← KV[i].key
19: KV2[local_nleft].value ← $KV[i]$.value
20: local_nleft ← local_nleft + 1
21: end if
22: for $i ←$ beg, end do
23: start ← atomicAdd(nleft, local_nleft)
24: KV[start].key ← KV2[i].key
25: KV[start].value ← $KV2[i]$.value
26: start ← start + 1
27: for $i ←$ 0, [local_table] − 1 do
28: if local_table[i] is claimed then
29: updateInsert(table, local_table[i].key,
30: local_table[i].value, update)
31: end if
32: end for
33: beg ← (nleft/|T|) * $t_i$
34: end ← (nleft/|T|) * (t_i + 1) - 1
35: for $i ←$ beg, end do
36: updateInsert(table, KV[i].key,
37: $KV[i]$.value, update)
38: end for
39: end for
40: end procedure
```
but the XMT’s performance also improved. The improvement ranges from around a 10% decrease in time for low processor counts, to nearly a 90% improvement for 128 processors, due to the greater scalability of the RHH strategy.

While performance improved significantly under RHH for both the XMT and the UV on power law distributions, it actually suffered under uniform random distributions. For example, the times for RHH_K increased on average by a factor of 49% on the UV, while the XMT incurred a 90% increase. This is due to the added computational complexity. Both are, of course, O(k|KV|), but the constant differs. We’ll omit a formal count on the number of operations, but we can perform a rough count on the number of iterations for a uniform random data set.

CO has one main for loop with |KV| iterations. RHH, on the other hand, has a parallel region with five for loops. The first for loop iterates over a sample of the data and requires f|KV| iterations. The second for loop iterates over the entire data set, equating to |KV| iterations. The third for loop iterates over all the key-value pairs not caught by the sampling. For the uniform random data set, which has virtually no duplicates, this means (1−f)|KV| iterations are required. The fourth for loop iterates over the size of the local table. We set our local tables to be of size 2^m, thus incurring 2f|KV| iterations. Finally, in the last loop, we iterate over the set of remaining key-value pairs, which comes again to (1−f)|KV|.

The reason RHH performs significantly better for power law distributions is due to the fact that RHH exploits the NUMA aspects of the machine: the cache hierarchy for the UV, and the data buffers of the XMT.

A simple compromise to get the best of both worlds would be to sample the data to understand the distribution. If it appears uniform random with no duplicate keys, the process can continue with CO, thus incurring |KV|(1 + f) iterations. If the data, by statistical measures, appears to have frequent keys and approaches a power law distribution, the process can continue with the RHH strategy. A simple heuristic that will likely work well in practice is the following: First find the mode from the sample, m₀, and estimate the probability of finding m₀ from a random selection within the array to be p(m₀) = |m₀|/|KV|, where |m₀| is the number times m₀ was found in the sample. Then we can estimate the number of times that t threads will collide on updating m₀ with the expected value of a binomial distribution, namely t·p(m₀). One can then find empirically a threshold, τ, such that for t·p(m₀) > τ, the RHH strategy is warranted.

Before concluding, we examine the spatial complexity of each of the approaches. Table 2 lists the amount of memory needed for each method. The variables k_v, l_v, and l_o are the size of the keys, values, and locks, respectively. For our experiments keys and values are 8-byte integers. Locks for the UV are 4-bytes, while the XMT requires 8-bytes. |table| is the length of the hash table. For uniform random data sets, the table size is on the order of |KV|, the length of the key-value array. For power law distributions, the table size can be much smaller as many keys repeat within the distribution. The 2f|KV| term in the RHH methods relates to the amount of space needed for the thread-dedicated tables. Each local table is of length 2f|KV|/|T|, where f is the sample fraction, |T| is the number of threads, and 2 is used to ensure appropriate load factors on the table. Thus, the length of all the local tables in their entirety comes to 2f|KV|. The last term |KV| is the buffer used to gather remaining keys together after frequent keys had been removed.

<table>
<thead>
<tr>
<th>Method</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>CO_K</td>
<td>k_v,l_v,</td>
</tr>
<tr>
<td>CO_U</td>
<td>k_v,l_v,</td>
</tr>
<tr>
<td>RHH_K</td>
<td>k_v,l_v,</td>
</tr>
<tr>
<td>RHH_U</td>
<td>k_v,l_v,</td>
</tr>
</tbody>
</table>

Table 2: This table displays the space requirements for each method.

6. CONCLUSIONS

The RHH strategy for hashing, either for known key distributions or unknown key distributions, scales well on both the XMT and the UV. Also, this scalability is exhibited for both uniform random and more difficult power law distributions.

For the size of problem we considered, that of five billion integers, the UV appears to be the clear winner, especially when we consider that only 4 Xeon processors were needed to topple the might of 128 Threadstorm processors. The RHH strategy effectively exploits the NUMA aspects of the machine, allowing the cache to remain stable for most of the computation. However, the XMT exhibits greater scalability and efficiency. We are curious if, for larger systems and problems, the XMT will eventually surpass the performance of the UV.

Acknowledgments

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy’s National Nuclear Security Administration under contract DE-AC04-94AL85000.

7. REFERENCES


