# Design and Evaluation of Parallel Hashing over Large-scale Data

Long Cheng[1,2,3], Spyros Kotoulas[2], Tomas E Ward[1], Georgios Theodoropoulos[4]

[1] National University of Ireland Maynooth, Ireland
[2] IBM Research, Ireland
[3] Technische Universität Dresden, Germany
[4] Durham University, UK
long.cheng@tu-dresden.de, spyros.kotoulas@ie.ibm.com, tomas.ward@nuim.ie, theogeorgios@gmail.com

***Abstract*—High-performance analytical data processing systems often run on servers with large amounts of memory. A common data structure used in such environment is the hash tables. This paper focuses on investigating efficient parallel hash algorithms for processing large-scale data. Currently, hash tables on distributed architectures are accessed one key at a time by local or remote threads while shared-memory approaches focus on accessing a single table with multiple threads. A relatively straightforward "bulk-operation" approach seems to have been neglected by researchers. In this work, using such a method, we propose a high-level parallel hashing framework, Structured Parallel Hashing, targeting efficiently processing massive data on distributed memory.**

**We present a theoretical analysis of the proposed method and describe the design of our hashing implementations. The evaluation reveals a very interesting result - the proposed straightforward method can vastly outperform distributed hashing methods and can even offer performance comparable with approaches based on shared memory supercomputers which use specialized hardware predicates. Moreover, we characterize the performance of our hash implementations through extensive experiments, thereby allowing system developers to make a more informed choice for their high-performance applications.**

***Keywords*-Hash tables; parallel hashing; distributed hash tables; thread-level parallel; high performance**

## I. Introduction

Hash tables are commonly used data structures to implement associative arrays. The $O(1)$ expected time for most critical operations puts them at a significant advantage to competing methods, especially for large problems. Hash tables are the dominant structure for applications that require efficient mappings, such as database indexing, object caching and string interning.

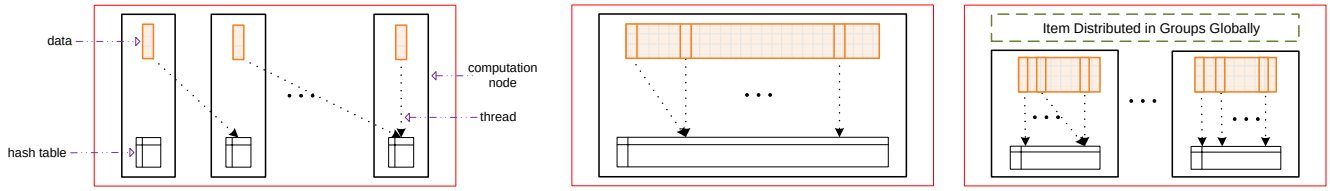
**State-of-the-art.** As applications grow in scale, parallel hashing on multiple CPUs and/or machines is becoming increasingly important. There are two dominant parallel hashing frameworks that are widely used and studied: distributed and thread-level parallel hashing.

For the first framework, as shown in Figure 1(a), the threads at each computation node (either logical or physical) build their own hash tables firstly, and then process the initial partitioned data (refer as *keys* for simplification throughout this work) through access a local or remote hash table(s). In general, this access is determined by hash values of the processed keys. This approach is very popular in distributed systems. Considering the target for high performance computing, in the following we only discuss the conditions of full parallelism, rather than the hash tables used in *peer-to-peer* systems, for example, the common studied *Distributed Hash Tables* (DHTs) [1].

In thread-level hashing, (Figure 1(b)), a single hash table is constructed on the single underlying platform, and multiple available threads operate with coordination on that table in parallel. This particular model is widely studied for multithreaded platforms which range in scale from commodity servers to supercomputers. As there exists no costly network communication (though possible NUMA) under this scheme, it always performs very fast.

The two parallel schemes scale in terms of processing large numbers of items by employing new nodes or threads. However, both approaches meet performance issues when processing massive data. With distributed hashing, the large number of frequent and irregular remote accesses of hash operations across computational nodes is costly in terms of communication. Moreover, when the processed data has significant skew, the performance of such parallel implementations will be dramatically decreased because all the popular keys will flood into a small number of nodes and cause hot spots. For parallel hashing on multithreaded architecture platforms, the cooperation between threads can efficiently balance the workloads, regardless, both for the skewed or non-skewed data, the associated scalability is bound by the limit on the number of threads available, the availability of specialized hardware predicates and possible memory contention. Furthermore, memory and I/O eventually also become bottlenecks at very large scale.

**Approach.** In general terms, the memory hierarchy of modern clusters consists of a distributed memory level (across nodes) and a shared memory level (multiple hardware threads/cores accessing the memory of a single node). We




(a) Distributed parallel hashing (b) Thread-level parallel hashing (c) Structured parallel hashing

Figure 1. Comparison of different parallel hashing frameworks.

are proposing a *structured parallel hashing* (SPH) framework (shown in Figure 1(c)) that blends distributed hashing and shared-memory hashing, divided into two phases: (1) items are grouped and distributed globally by each thread, and (2) hash tables are constructed on each node and each of them is only accessed by a local thread(s).

The primary idea is a straightforward *bulk-operation* scheme, however, in so far as the authors are aware such a kind of hashing approach has not been previously described in the literature. Intuitively, this method has two advantages: (a) reduce remote memory access, load imbalancing and the associated time-cost arising from memory allocation, table locks and communication in distributed hashing, and (b) support high scalability compared to thread-level hashing (there are no hardware limitations as our approach operates using predicates available on all platforms).

In fact, such bulk operations are widely applicable. For example, in previous work [2] [3], we have implemented joins for parallel data processing using a similar approach. Namely, tuples of an input relation are redistributed to all the computation nodes. From that basis, local hash tables are created for lookup conducted by the other relation. In such application scenarios, the following three questions arising from the proposed framework are becoming to be interesting:

- performance: *will the responsible implementations be scalable and can they achieve comparable performance or even outperform the other two approaches?*
- parallelism: *how will the performance change with varying the number of threads over each hash table, if the whole available threads are fixed for a given system?*
- impact factors: *how will the high-level data distribution as well as the underlying hash table designs impact on the performance?*

The responsible answers will give us an insight of the underlying hash implementation as well as an option to further improve the performance of applications using hash tables over distributed memory.

**Contribution.** Motivated by our application domain as mentioned above, we propose a simple high-level parallel hashing framework, *structured parallel hashing*, targeting efficient processing of massive data on distributed memory. We conduct a theoretical analysis of this scheme and present an efficient parallel hashing algorithm based on it. We evaluate on an experimental configuration consisting of up to 192 cores (16 nodes) and large datasets of up to 16 billion items. The experimental results demonstrate that our approach is efficient and scalable. It is orders of magnitude faster than the conventional distributed hashing methods, and also achieves comparable performance with a shared memory supercomputer-based approach, on a socket-for-socket basis. Additionally, for the underlying hash tables, the proposed *range* lock-free strategy tailored for our framework is demonstrated to be faster than the conventional *compare-and-swap* operations.

The rest of this paper is organized as follows: In Section II, we conduct a theoretical analysis of different hashing frameworks. We present an efficient parallel hashing algorithm in Section III. In Section IV, we experimentally evaluate our work, followed by a comparison to the literature in Section V and our conclusions in Section VI.

## II. Theoretical Analysis of Hashing Frameworks

In our theoretical analysis, we make four assumptions: (1) our hash function produces a uniform distribution, (2) slot accesses after a hash collision follow a uniform random distribution, (3) each node can communicate with multiple remote nodes at the same time, and (4) the memory access and data transfer inside a physical node is zero (compared to the network-based operations). The first two assumptions are popular in currently theoretical studies [4] and the latter two are natural for an ideal distributed system. In addition to this, we refer to the distributed and thread-level hashing frameworks as HF1 and HF2 respectively, and our structured parallel hashing framework as HF3.

In general, the total time cost $T$ to insert $N$ items in a framework can be divided into three parts: distribution time for item transfers across memory resident in different nodes $t_m$, time for probing $t_p$ and time costs due to memory contention $t_c$ . As threads work in parallel in each framework, $T$ would be the same as the time $t$ by a single thread (assuming equal load). Specifically, we have $t_m = 0$ for HF2 as there is only a single shared memory location.

### A. Distribution

We assume that the time cost of moving an item to the node itself is 0, and the time $t(s)$ to transfer $s$ items to a remote node is $t(s) = \delta_0 + \delta_1 \cdot s$, where $\delta_0$ is a constant that represents the latency for each data transfer[1] while $\delta_1$ is the time for transferring a single item.

In a cluster with $n$ physical nodes in which each has a constant number of threads $e$, there will be $ne$ hash tables in HF1 and $n$ in HF3, and each thread will process $N/ne$ items. Since the items are processed one by one in HF1, the number of item transfers will be $N/ne$. In HF3, items are grouped into $n$ chunks by each thread (namely total $ne \cdot n$ chunks with $N/n^2e$ items each) and moved to the corresponding $n$ nodes. Since the ratio of moved items to a remote node is $(n-1)/n$, the item transfer time in HF1 and HF3 is:

$$t_{m_1} = \frac{n-1}{n} \cdot \frac{N}{ne} \cdot (\delta_0 + \delta_1) \tag{1}$$

$$t_{m_3} = \frac{n-1}{n} \cdot n \cdot (\delta_0 + \delta_1 \cdot \frac{N}{n^2 e}) \tag{2}$$

This indicates that: (1) if $n$ is a constant, $t$ will be $O(N)$, and (2) for a given $N$, $t$ will be $O(n)$. Additionally, if $n$ is fixed, the time difference $(t_{m_1} - t_{m_3})$ between HF1 and HF3 will be $O(N)$. It means that *with the increment of $N$, HF3 will spend less time on item transfers than HF1.*

### B. Slot Probing

In each framework, threads insert items using a pseudo-random probe sequence. For a successful insertion, the last probed slot is empty, while the slot accessed before (if any) is an occupied. For a hash table with $c$ slots and $v$ elements (load factor at end of execution $\alpha = v/c \leq 1$), according to the theorem for standard hashing presented in [4], we have the function between the average number of probes $l$ in a successful search and $v$:

$$l(v) = \frac{1}{\alpha} \sum_{i=c-v+1}^{c} \frac{1}{i} \tag{3}$$

HF1 and HF3 implement insertion on individual partitions of distributed memory. Therefore, we have $v_1 = N/ne$ and $v_3 = N/n$. Moreover, for the single node with $e'$ threads in HF2, there exists $v_2 = N$ and each thread processes $N/e'$ items. Normally, we have $l(v_1) = l(v_2) = l(v_3) = l_0 \approx (-1/\alpha) \cdot \ln(1-\alpha)$, because $N$ is a great number (for example 16 billions in our experiments) and there is $N >> ne$. If the time for a single probing operation is $\eta_0$, equal in each framework, then with the same load factor $\alpha$, the probing time for a single thread would be:

$$t_{p_1} = t_{p_3} = \eta_0 l_0 \cdot \frac{N}{ne} \tag{4}$$

[1]Note that connections for data transfer could be retained, regardless, extra time cost for remote accesses still exist, such as memory allocation etc.

$$t_{p_2} = \eta_0 l_0 \cdot \frac{N}{e'} \tag{5}$$

This implies that for a given underlying platform, the probing time of each framework will be $O(N)$. And *for a fixed input, HF1 and HF3 can reduce the probing time by increasing the number of nodes $n$.*

### C. Memory Contention

We define a *conflict* as the situation where more than one thread try to access the same hash table slot at the same time. The probability that a thread accesses a specified slot of a hash table ($c$ slots and $v$ elements) is $1/c$. With $w$ threads, the probability that $i$ ($1 \leq i \leq w$) threads access the same slot would be:

$$p(v,i) = \binom{w}{i} \left(\frac{1}{c}\right)^i \left(1 - \frac{1}{c}\right)^{w-i} \tag{6}$$

There will be $i-1$ thread conflicts when $i$ threads access a same slot. Under the condition that $w << v$, the average number of conflicts for probe operations for a thread would be:

$$\begin{aligned} c(v,w) =& \sum_{i=1}^{w} (i-1)p(v,i) \\ =& p(v,2) + \sum_{i=3}^{w} (i-1)p(v,i) \\ \approx& p(v,2) \approx \frac{w(w-1)\alpha^2}{2v^2} \end{aligned} \tag{7}$$

For uniformly expressing the cost of the three approaches, here we refer to the number of items processed by each thread as $h_k v_k$, where the subscript $k$ means the identify of each framework, namely $k = 1, 2, 3$. Then, we have $h_1 = 1$, $h_2 = 1/e'$ and $h_3 = 1/e$, which are all constant. If we assume that the waiting time $\lambda_0$ resulting from a single conflict in each framework is the same and there are $\bar{w_k}$ threads accessing a hash table, then, with the average number of probings described previously, we have:

$$t_{c_k} = \lambda_0 \cdot l_0 h_k v_k \cdot c(v_k, \bar{w_k}) = \frac{\lambda_0 l_0 \alpha^2 h_k}{2} \cdot \frac{\bar{w_k}(\bar{w_k} - 1)}{v_k} \tag{8}$$

With a limited[2] $n$, $e$ and $e'$, $\lambda_0 l_0 \alpha^2 h_k \bar{w_k}(\bar{w_k} - 1)/2$ will be a limited constant. Because $v_k$ is $O(N)$, the time $t_{c_k}$ will be $o(1/N)$. It means that *when processing very large-scale data, the time cost for memory contention can even be neglected in all frameworks.*

[2]For example, for the cluster we use in our experiments, there is $n = 16$ and $e = 12$. For a supercomputer, the $e'$ could be hundreds or thousands.

### D. Performance Comparison

When processing large data ($t_{c_k} = 0$), the time difference $\Delta T_{ij} = \Delta T_i - \Delta T_j$ between HF3, HF2 and HF1 is:

$$\Delta T_{13} = \delta_0 \cdot (n-1) \cdot (\frac{N}{n^2 e} - 1) \tag{9}$$

$$\Delta T_{23} = \eta_0 l_0 \cdot (\frac{N}{e'} - \frac{N}{ne}) - (n-1) \cdot (\delta_0 + \delta_1 \cdot \frac{N}{n^2 e}) \tag{10}$$

With a limited $n$, $e$ and $e' \geq 2e$, if we set $N \to \infty$, then we have: (1) there is always $\Delta T_{13} > 0$ and (2) let $k_1 = \eta_0 l_0 / e$ and $k_2 = k_1^2 + (\delta_1/e)^2 + (2 - 4e/e') \cdot k_1 \delta_1 / e$, there will be $\Delta T_{23} > 0$ when

$$n \geq \frac{e'}{2k_1 e} \cdot (k_1 + \frac{\delta_1}{e} + \sqrt{k_2}) \tag{11}$$

This implies that *when processing a very large data set, (i) our hash framework is always faster than HF1, and (ii) it can perform better than HF3 with increasing the number of computation nodes*, at least based on a high-level theoretical analysis and on a simplified model. This assertion will be tested in the experimental evaluation.

## III. Parallel Hashing

In this section, we present an efficient parallel hashing algorithm based on our framework. We focus on techniques to (1) maintain consistency in the distribution phase, and (2) avoid hash collisions and memory contention during hash operations. Additionally, motivated by the performance of data storage and information lookups in our applications [2] [3] [5] [6] [7], we just focus on the hash operations of insertion and searching.

### A. Distribution

For an $n$-node system and $t$ threads per node, all the threads read and distribute items in parallel. We introduce an integer parameter $i$ to subdivide items based on a common $(n \times t)$-based hash partitioning, namely set $h(key) = key \ mod \ |n \times t \times i|$ to group and distribute items, based on the hash values of their *key*. Then, groups with hash values in the range $[k \cdot (t \cdot i), (k+1) \cdot (t \cdot i)]$, are sent to the $k$-th node ($k \in [0, n-1]$).

After the distribution, each computation node (rather than thread) has total $t \times n \times (t \times i)$ chunks of data to be processed locally. We treat all of them as a data **cuboid** where each chunk is indexed by $(t, m, n)$, which represents that the chunk comes from the $t$-th thread with the hash value $m$ at the $n$-th node. The detailed implementation at each node is given in Algorithm 1. The array *item_c* is used to collect the grouped items, and its size is initialized by the number of thread $t$ and the modulo value $m$. Since each thread manages its own items, the reading and distribution operations can be performed in parallel across threads.

It is obvious that, for a given input dataset, the size of the cuboid (which is proportional to the number of received items for even data distributions) at each node will be constant, no matter how large the parameter $i$ is. Nevertheless, with a larger $i$, the chunks of the cuboid would be more fine-grained. This means that the size of the transferred data as well as the allocated memory each time will be smaller. Furthermore, if we process the cuboid data at the unit of a chunk, the workload of each local thread would also be more uniform. However, inter-node communication will become more frequent, negatively affecting performance. We will examine this trade-off in our evaluation.

**Algorithm 1** Item Distribution at each node

```
1: Initialize items_c:array[array[array[item](m)]](t)
2: for i ∈ threads async do
3:     Read in item file f
4:     for item ∈ f do
5:         des ← hash(item.key)
6:         items_c(i)(des).add(item)
7:     end for
8:     for j ← 0..(m − 1) do
9:         Push items_c(i)(j) to r_items_c(i, j, k) at node
       k, where k = m/n
10:    end for
11: end for
```

### B. Processing

Since the number of received items at each node can be easily recorded in the distribution phase, we can directly allocate the required size hash table and initialize it. In the meantime, during key insertion, there exist various hashing strategies to minimize hash collisions and different mechanisms like the lock-based and non-blocking approaches are proposed to address the problem of memory contention [4] [8] [9]. As we focus on the parallelism over hash tables, we adopt *linear probing* for hash collisions and *CAS* (compare-and-swap) for memory contention in our implementations, which is very popular in recently studies [10] [11] [9] [12] [13] [14]. In addition, we propose a new *range-based* algorithm, aimed at removing memory contention.

*1) CAS:* Compare-and-swap ensures the slot for the key that it is about to insert does not have another key inserted during its operation [12]. As shown in Figure 2, the hash table is initialized by two arrays. One array is used to hold the items, and other one is a status array using CAS to indicate whether the corresponding slot in the former array is filled or not. In the operation of insertion, the slot of an item is located by its hash value. The thread first checks whether that slot in the item array is filled or not. If not, the thread would atomically check and set the slot at the same index of the status array. If the slot is already set, the thread will continue to the next slot.

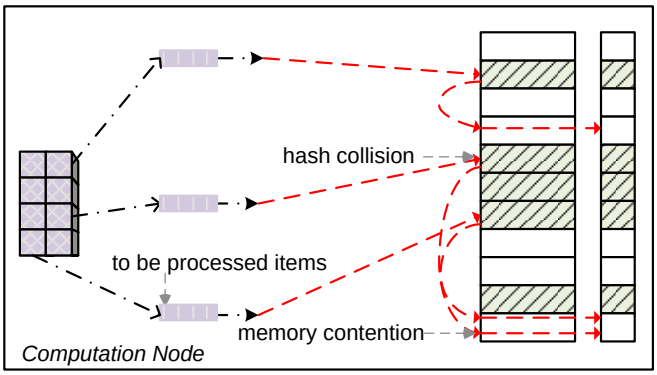



Figure 2. CAS-based Implementation.

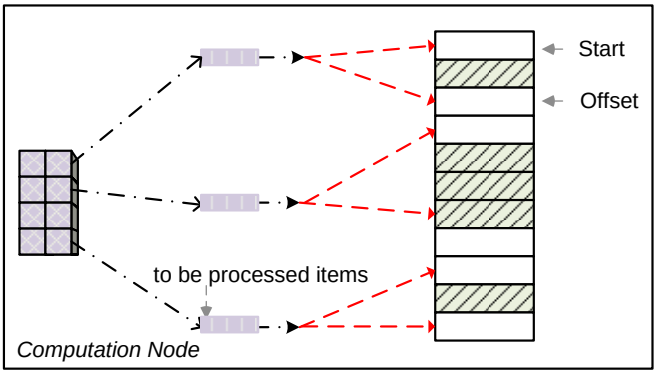



Figure 3. Range-based Implementation.

**Algorithm 2** CAS-based Implementation at each node

```
1: Allocate buffers for hash table ht:array[item],
     stat:array[atomicBoolean](true),
2: for each r_items_c(i, j, k) async do
3:     for item ∈ r_items_c(i, j, k) do
4:         e ← h1(item.key)
5:         Search an empty slot start from the eth position
6:         if ht(e).null ∧ stat(e).CAS(true,false) then
7:             ht(e) ← item
8:         else
9:             e + +, Continue searching
10:        end if
11:    end for
12: end for
```

The details of our implementation is shown in Algorithm 2. We use the array *ht* to store the items while the array *stat* is used to indicate occupancy. Each slot in the stat array is initialized with the element `atomicBoolean` to support the CAS operations. After that, the received data chunks will be scheduled as a task queue and assigned to all the available threads. For each item, the initial location of the slot will be calculated by a hash function $h_1(k)$. The empty slot searching process will start with the position $h_1(k)$ of $ht$. If a slot is not occupied and the CAS operation over $stat$ also returns the value of *true*, then the item will be inserted, otherwise, the next slot will be probed. We use a modular arithmetic to cycle the location of an array from the bottom to the top and the searching process will be repeated until a free slot is found. The insertion progress will be ended when all the inserting tasks are finished.

There are two possible issues when using CAS: (1) the *ABA* problem as described in [9]. Because the key value of a slot in our implementation only changes from *null* to another value and never changes back again, the same as the scenarios in [12], therefore the ABA problem could not exist in our implementations; (2) the *contention hot spots* problem as presented in [14]. In fact, this problem becomes a performance issue because [14] focuses on the study of continuously changing the same variable with multi-threads. In contrast, threads in our method do not work on a specified slot but over the whole table instead. From the probability as we analyzed in Section II, it is clear that the performance of our implementations will be not affected by such an issue, at least for the massive uniform distributed data.

*2) Range:* We propose the *range-based* approach from the basis of the *parallel radix join* [15] algorithm, which is commonly used in recently research with target for efficient parallel joins [15] [16]. The main idea is that the subdivided data is assigned to individual threads and then each thread processes the data independently. Regardless, the method for *joins* focuses on workload assignment in hardware-level, such as that the size of data chunks is set to the cache size so as to minimize the cache miss etc. Compared to that, our approach is concentrated on that all the threads can work on a given hash table without any influence by each other.

In general, as demonstrated in Figure 3, we map chunks of data in the cuboid to the specified hash table according to the value of index $m$ as we described previously. Because we can easily calculate the size of the mapped chunks, the mapped *range* on a hash table can be simply presented by two values: the start point *start* and the size *offset*. If threads at each node process the items in the unit of chunk section (according to index $m$), then all of them would work in a specified range, and no memory contention happens.

The detailed implementation at each node is presented in Algorithm 3. We first compute the size of each range by $S_k$=$\sum_{i,j}$ $item\_c(i, j, k).size$. When inserting an item, three parameters - the item, the start slot $R_{k-1}$ and the end slot $R_k$ of the range, are transferred to the hash function $h_2$ to locate the start probing point in the hash table. Similarly, we also use a modular arithmetic to ensure that the probings work in the specified range. The program is terminated when all the places finish item insertion.

The calculation of each *range* depends on the distribution of values in the hashtables, which can be easily computed in our data distribution phase. Therefore, the proposed

**Algorithm 3** Range-based Implementation at each node

```
1: Compute the kth Range:S_r
2: The end slot of the kth Range R_k=∑_{r<k} S_r
3: Allocate buffers for hash table: ht:array[item]
4: for each items_c(i,j,k) async do
5:     for item ∈ items_c(i,j,k) do
6:         e ← h_2(item.key, R_{k-1}, R_k)
7:         Searching a empty slot start from the eth slot
8:         if ht(e).null then
9:             ht(e) ← item
10:        else
11:            e++, Continue searching in [R_{k-1}, R_k]
12:        end if
13:    end for
14: end for
```

*range* method fits our framework well. We will evaluate its performance and compare it with the popular CAS method.

*3) Searching:* Searching is very similar to insertion. Keys are mapped to locations in the same way as in Section III-A and threads can independently search on the hash table. Because no thread synchronization is required in this phase (assuming no concurrent writes), threads in the CAS-based implementations can freely access any slot without checking the status array. For the range-based implementation, an additional operation is required to read the range.

## IV. Evaluation

**Platform.** Our evaluation platform is the *High-Performance Systems Research Cluster* located at IBM Research Ireland. Each computation unit of this cluster is an iDataPlex node with two 6-core Intel Xeon X5679 processors running at 2.93 GHz, resulting in a total of 12 cores per physical node. Each node has 128GB of RAM and a single 1TB SATA hard-drive and nodes are connected by Gigabit Ethernet. We implement our algorithms with the parallel language X10 [17] over the RHEL with Linux kernel 2.6.32-220. We use X10 version 2.3 and compiling it to C++ over gcc version 4.4.6.

**Dataset and Metric.** Table I shows the input and output parameters for our experiments, with bold font indicating default values. We have generated several datasets up to 16 billion integers. Data follows a uniform distribution when *Zipf factor* is equal to 0, or a skewed distribution with the associated $\alpha$ parameter. We mainly measure the runtime of each test in terms of: *distribution time*, *insertion time*, *hashing time* and *search time* as described. In the meantime, two types of hash tables based on our framework are examined: (1) *Structured Distributed Hash Tables (`SDHT`)*, in which there is a single thread per logical computation node. Therefore, this kind of hash table does not suffer from memory contention, but at the cost of reduced flexibility in terms of load balancing. (2) *Hybrid Parallel Hash Tables (`HPHT`)* have multiple threads per logical node operating with the CAS or *range* strategies (referred to as Range in the following) as described before.

Table I
Experimental Parameters.

*Input Parameters*

| Parameter | Values |
|---|---|
| Hash table implementation | `SDHT`, **`HPHT`**, CO_U, RHH_U |
| Dataset size (billions) | 0.5, **1**, 2, 4, 8, 16 |
| Zipf factor | **0**, 0.2, 1, 1.8 |
| Load factor | 0.6, **0.75**, 0.9 |
| #Threads | 12, 24, 48, 72, 96, 120, 144, 168, **192** |
| #Threads/Table | **4**, 12 |
| #Threads/Core | **1** |
| Hash Collision Strategy | CAS, **Range** |
| $i$ parameter | **1**, 10, 100 |
| Key length | 32 bits, **64 bits** |

*Output Parameters*

| Parameter | Description |
|---|---|
| Read time | Time to read data from disk |
| Distribution time | Time to distribute items |
| Insertion time | Time to insert to hash-table |
| Hashing time | Sum of Distribution and Insertion time |
| Search time | Time to search for all items |

In the following, we first conduct the performance comparison of each hash framework on a basic test. Then, we evaluate the scalability of `SDHT` and compare the performance of `HPHT` using different lock-free strategies. Finally, we study the impact factors of our hash tables and compare our results with current implementations as presented in [18] and [10]. Because the standard deviation between executions was very small in our tests, we record the mean value based on ten measurements.

### A. Comparison of Frameworks

We conduct a simple performance comparison of the three hashing frameworks already described based on the CAS strategy. We implement the thread-level parallel on a single machine with 12 node, and other two parallelism on a distribute system with the same node, but just use 6 cores each machine (namely a small machine). We process 10 million keys and present the result in Figure 4. There, the configure $2 \times 6$ indicates a configuration of two machines using 6 cores each. It can be seen that our framework *HF3* performs much better than *HF1*. However we are slower than *HF2* initially but when using 4 machines (24 cores), our implementation become faster. All this is consistent with our theoretical analysis in Section II.

### B. Structured Distributed Hash Tables

We test the scalability of our *SDHT* by varying the number of processing threads and the size of input data.

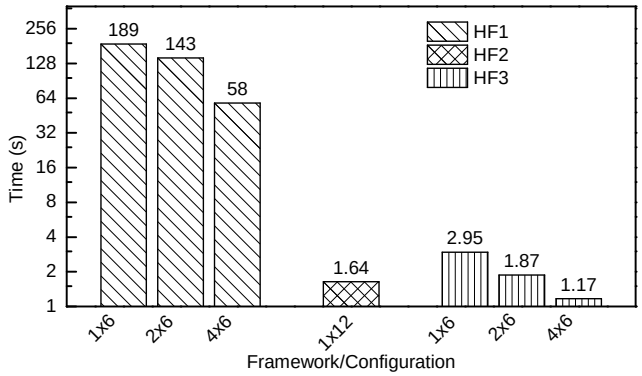



Figure 4. Performance comparison of three frameworks.

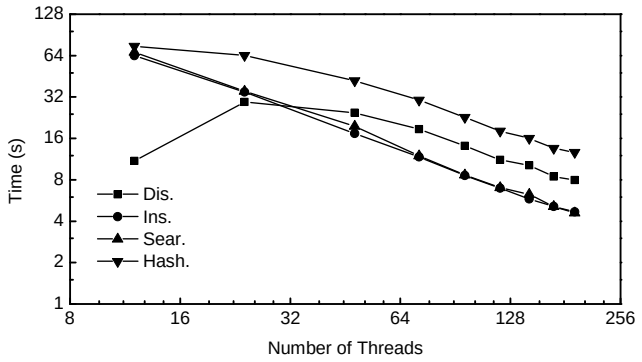



Figure 5. Time cost with varying number of threads for SDHT.

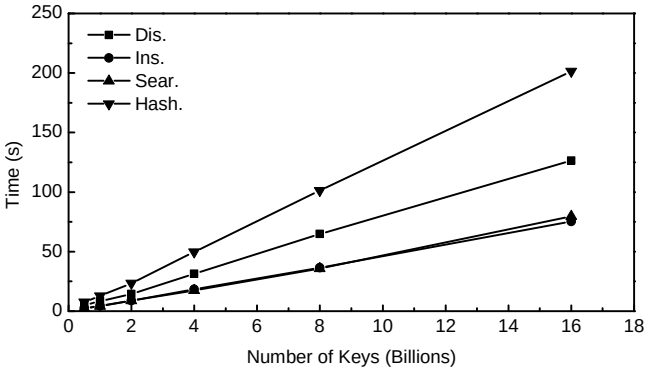



Figure 6. Time cost with varying size of input for SDHT.

The results are shown in Figure 5. We can see that the time cost for inserting and searching is almost the same and linear with the number of nodes. For a small number of threads, distribution time is not linear, since for a single node there is no network communication and for two nodes (24 threads), only 50% of the data needs to be transferred over the network. With more than 72 threads, the distribution cost decreases linearly with the number of nodes. Overall, the hashing time follows the same pattern.

To study the scalability of our algorithm with increasing input size, we fix the number of threads to 192 (16 nodes), start our tests with 500 million integers and repeatedly double the size of the input until 16 billion. The results are presented in Figure 6. The hashing time is linear with the size of the input, and nearly matches the ideal speedup scenario. The same holds for distribution, insertion and searching. Furthermore, the time spent in the insertion phase is nearly the same as the searching phase, and both are less than that of the distribution phase.

From the results above, we can see that hash table construction scales very well both with the number of threads and the input size. We also notice that, with a 16-node cluster, the item distribution costs about 60% more than insertion and searching. We will characterize the possible factors in Section IV-D.

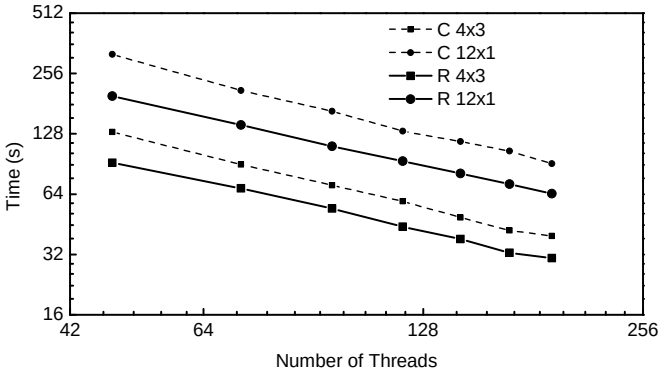



Figure 7. Time cost with varying the number of threads for HPHT.

### C. Hybrid Parallel Hash Tables

We elaborate on the performance of HPHT for different strategies and input parameters. HPHT uses multiple threads per node (could be logical or physical), and by extension, multiple hashtables. We choose two typical cases: four threads and three *logical nodes* per physical node ($4 \times 3$) and twelve threads and one *logical node* per physical node ($12 \times 1$). We further experiment with regard to scalability with the number of threads.

Figure 7 presents runtimes to process 1 billion integers. Similar to SDHT, both the CAS and Range implementations scale well with the number of threads. With detailed runtime comparison, we find that the proposed *Range* method performs much faster than CAS, both for insertion and searching. There are three possible reasons: (1) hash table construction in CAS is more complex (using extra-arrays); (2) there is extra atomic compare-and-swap operations in CAS while there is no memory contention in Range; and (3) regarding search, although there are no compare-and-swap operations for CAS, Range still benefits from superior memory locality for individual threads.

Given a fixed number of threads, the implementation configured with $12 \times 1$ performs worse than $4 \times 3$ for

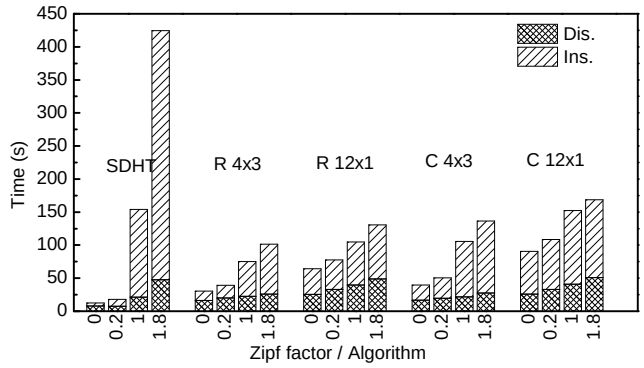



Figure 8. Runtime by varying *Zipfian factor* in each implementation.

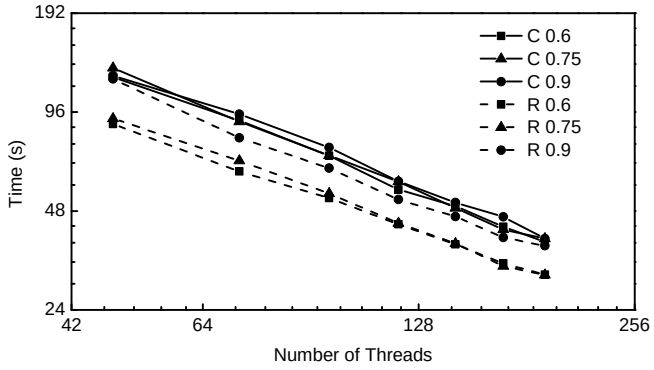



Figure 9. Time cost with different load factors.

each algorithm, and both of them are slower than SDHT. Although HPHT is slower than SDHT when processing uniformly distributed integers, HPHT scales equally well. Moreover, HPHT features thread coordination, which would be advantageous in some scenarios, such as against the data skew.

To validate this claim, we conduct a test on 16 nodes under dataset skew. Each dataset contains 1 billion integers following the Zipf distribution ($\alpha = 0.2$, 1 and 1.8). To support the thread coordination in Range operations, we also set the parameter $i$ to 10 in each implementation (recall that threads in Range operations process data chunks according to the value of modulo, so there will be no thread coordination if $i = 1$). As shown in Figure 8, distribution time and insertion time increases with the skew of the dataset for all settings. However, for $a \geq 1$, HPHT significantly outperforms SDHT, indicating superior load balancing, mainly during insertion. Additionally, the configuration with $12 \times 1$ is still slower than $4 \times 3$, which means that hash tables with moderate parallelism could be a better choice even in the presence of high skewed data.

### D. Impact Factors

We also test factors that with potential performance impact. As the curves with different numbers of threads per place are nearly the same, both in SDHT and HPHT, we only present results with the configuration $4 \times 3$ based on CAS and Range implementations. The factors we have considered are (a) the *load factors* of hash tables, (b) the *length of the processed keys* and (c) the parameter $i$, mentioned in Section III-A.

Three different values for load factor (0.6, 0.75 and 0.9) are examined in our tests. Figure 9 shows the hashing time for 1 billion integers. Once again, we observe that all the implementations scale well with the number of threads. In the meantime, as expected, hashing time increases with the load factor. For both strategies, the runtime with load factor 0.6 and 0.75 is nearly the same. For a load factor of 0.9, in Range, runtime increases by nearly 20% while for CAS, it increases only by 3%. This also indicates that hash collisions have a more significant effect on the performance for the Range implementation. There is a trade-off between the memory consumption and the load factor, therefore, in real implementations, assigning the load factor to 0.75 would be a better choice for Range and 0.9 if using CAS.

Table II
DETAILED TIME COST OF PROCESSING DIFFERENT INTEGER LENGTHS

| # Threads | 64-bit integer (sec.) | | | 32-bit integer (sec.) | | |
|---|---|---|---|---|---|---|
| | Dis. | Ins. | Sear. | Dis. | Ins. | Sear. |
| 48 | 38.63 | 53.05 | 55.65 | 18.26 | 55.39 | 56.91 |
| 72 | 29.02 | 39.28 | 38.26 | 15.03 | 37.44 | 36.24 |
| 96 | 25.51 | 28.80 | 28.77 | 13.22 | 27.66 | 28.94 |
| 120 | 21.23 | 22.84 | 22.78 | 11.34 | 22.27 | 24.01 |
| 144 | 19.78 | 18.46 | 20.10 | 10.47 | 18.61 | 18.92 |
| 168 | 16.37 | 16.26 | 16.74 | 9.33 | 16.00 | 17.01 |
| 192 | 16.40 | 14.26 | 14.19 | 9.98 | 14.12 | 15.01 |

We test the time of processing 1 billion integers represented with 32 bits or 64 bits. Because the CAS and Range implementations show the same characteristics, we only present the execution time for the Range algorithm as shown in Figure II. The time spent on distributing the 32-bit integers is about a half of that for the 64-bit objects, while the insertion and the search time do not change. This is in contrast with the conclusion in [18] that varying the size of integers has no effect on time. This difference shows the essential difference between our implementation and other general algorithms: we used a high-level structured method to group items that need to be sent to the remote nodes, while other methods send many short messages that overwhelm the network, leading to significant inter-node communication and coordination overhead. This can also be observed in our results in that the distribution with 168 threads and 192 threads takes nearly the same time, because the transferred data chunks become too small.

Finally, we evaluate how the data partitioning in the distribution phase affects the execution time. The parameter $i$ is set to 1, 10 and 100 respectively and the results are present in Figure 10. The runtime in both CAS and Range

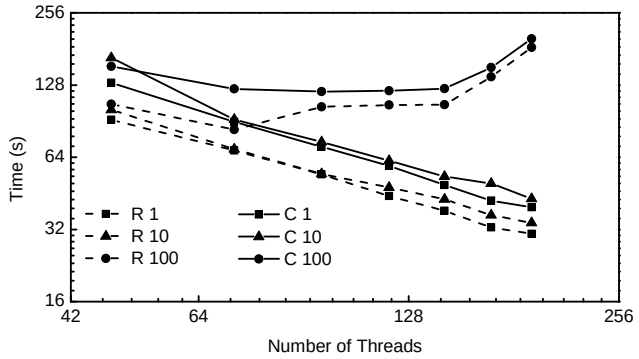



Figure 10. Time cost with varying the parameter $i$.

with $i = 10$ is slightly greater than that with $i = 1$, and is fairly linear with the number of threads. However, when setting the parameter to 100, the time cost decreases at first and then increases with the number of threads, leading to bad scalability. The decrease in the size of transferred data for each thread at the beginning reduces the distribution time, but as the number of threads increases, the vastly increased number of chunks incurs significant coordination overhead. The above result, together with our experiments regarding skew, indicate that higher $i$ should be chosen for larger data sizes and higher skew.

### *E. Comparison with Current Implementations*

The latest evolution with *distributed parallel hashing* is reported in [18], using 768 threads on a cluster to process 19.2 million items takes 18.2 secs using UPC and 27.4 secs using MPI. In comparison, our best performing implementation can process 1 billion items in just 13 secs with 192 threads, and we also achieve linear scale with the increment of threads. This is similar as the results presented in Figure 4, and the great difference evident arises from the different hashing frameworks utilised.

We also conduct a detailed comparison with the fastest performing implementation in the literature, presented in [10]. [10] implements *thread-level parallel hashing* on a Cray XMT supercomputer using two techniques: CO and RHH[3]. Although the approach in [10] also optimizes hashing of skewed loads, it is not the focus of this paper. Figure III shows the file reading and hashing time to process **5 billion** integers. The Cray XMT is a shared-memory architecture using a specialized interconnect and a latency-tolerant model. Since a direct comparison of processor speeds in not meaningful, we group the results on a per-socket basis. We observe that, `SDHT` is faster than RHH and slower than CO if we do not consider the reading time. If we consider reading time, `SDHT` is faster than all other techniques and systems. `HPHT` is slower than all other approaches, but still

[3]the results presented here were obtained by communication with the authors

Table III
COMPARISON WITH RESULTS PRESENTED IN [10] (TIME IN SECONDS)

| Algorithm | 16 Sockets | | 32 Sockets | |
|---|---|---|---|---|
| | Read. | Hash. | Read. | Hash. |
| Cray CO_U | 123 | 90 | 123 | 46 |
| Cray RHH_U | 124 | 150 | 123 | 77 |
| `SDHT` | 57 | 113 | 30 | 59 |
| Range $4 \times 3$ | 70 | 257 | 32 | 152 |
| Range $12 \times 1$ | 72 | 570 | 36 | 301 |
| CAS $4 \times 3$ | 68 | 331 | 32 | 192 |
| CAS $12 \times 1$ | 72 | 842 | 37 | 466 |

remains within an order of magnitude of the best performing system. Overall, although our system relies exclusively on low-cost commodity hardware, we observe that it achieves comparable performance to a shared-memory system using a specialized interconnect and processor architecture. With increasing nodes, it is expected that we can even outperform [10] on the *hash* operation on the basis of the theoretical analysis in Section II.

## V. RELATED WORK

The study of distributed parallel hashing main focuses on (1) low-level communication schemes such as the use of the IBM LAPI [19], and (2) parallel programming paradigms/languages, such as the use of Java, MPI and UPC [20] [21] [18]. In these implementations, hash operations are always accompanied with frequent and irregular remote memory access with a concomitant increase in low-level communication overhead and the associated performance hit. Therefore, they are more suitable for processing small data, but not for massive data.

There is long history of theoretical studies [22] [23] in terms of the thread-level parallel approaches. By employing different hashing strategies, implementations on various platforms have achieved excellent performance [12] [11] [10]. Our implementation performs comparably or slightly worse than the fastest one [10], however our approach relies on low cost commodity hardware, adding to its flexibility.

GPU computing has become a well-accepted parallel computing paradigm and there are many reports on implementations of parallel hashing based on that [24] [25]. Implementations of these hash tables exhibit strong performance. However, GPU memory is limited so therefore such methods cannot work with excessively hash tables of the sizes shown in this paper. In addition, reading data into GPUs takes a considerable time, adding significant overhead for a simple task, from the perspective of computation.

Although *parallel hash joins* are widely studied in modern parallel database management systems [15] [16] [26], there is little research focuses on the parallelism of underlying hash tables. With the increase in size of process datasets in

this domain [26], we expect that the hash strategies used in our hash tables can further improve join performance here.

The idea behind our method is straightforward, yet not trivial, and does not appear in the literature. Consequently we believe that the evaluations conducted here and the results described are of value to the community as a basis for understanding the merits of the approach. Moreover, our theoretical analysis in Section II confirm that our structured method is faster for large datasets - a result verified through our experiments. Finally we also contribute a *range-based* strategy for our hashing implementation, which is shown to be faster than the commonly used *CAS* method within our framework.

## VI. Conclusions

In this work, we proposed a high-level structured framework for parallel hashing, which has been designed for processing massive data. This framework supports (a) distributed memory while avoiding frequent remote memory access, and (b) thread coordination on a per-partition basis. Based on that, we presented an efficient parallel hashing algorithm by employing the popular CAS and our proposed *range-based* lock-free hashing strategies.

The experimental evaluation results show that our implementation is highly efficient and scalable in processing large datasets. Additionally, this hash framework demonstrates useful flexibility in that it can employ various hashing techniques and can be run on commodity hardware. Finally, the proposed Range lock-free strategy is faster than the conventional CAS operation and presents better load balancing characteristics than approaches which use a single thread per partition.

Our future work lies in extending our current approach with methods to better handle skew and applications in large and robust parallel joins.

## Acknowledgments

This work is supported by the Irish Research Council and IBM Research Ireland.

## References

[1] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 17–32, Feb. 2003.

[2] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos, "Efficiently handling skew in outer joins on distributed systems," in *CCGrid*, 2014, pp. 295–304.

[3] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos, "Robust and efficient large-large table outer joins on distributed infrastructures," in *Euro-Par*, 2014, pp. 258–269.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2001.

[5] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos, "QbDJ: A novel framework for handling skew in parallel join processing on distributed memory," in *HPCC*, 2013, pp. 1519–1527.

[6] L. Cheng, A. Malik, S. Kotoulas, T. E. Ward, and G. Theodoropoulos, "Efficient parallel dictionary encoding for RDF data," in *WebDB*, 2014.

[7] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos, "Robust skew-resistant parallel joins in shared-nothing systems," in *CIKM*, 2014.

[8] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, Jan. 1991.

[9] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Elsevier Science, 2011.

[10] E. Goodman, M. N. Lemaster, and E. Jimenez, "Scalable hashing for shared memory supercomputers," in *SC*, 2011, pp. 41:1–41:11.

[11] E. Goodman, D. Haglin, C. Scherrer, D. Chavarria-Miranda, J. Mogill, and J. Feo, "Hashing strategies for the Cray XMT," in *IPDPS Workshop*, 2010, pp. 1–8.

[12] A. Stivala, P. J. Stuckey, M. Garcia de la Banda, M. Hermenegildo, and A. Wirth, "Lock-free parallel dynamic programming," *J. Parallel Distrib. Comput.*, vol. 70, no. 8, pp. 839–848, Aug. 2010.

[13] D. Zhang and P.-A. Larson, "LHlf: lock-free linear hashing," in *PPoPP*, 2012, pp. 307–308.

[14] D. Dice, D. Hendler, and I. Mirsky, "Lightweight contention management for efficient compare-and-swap operations," in *Euro-Par*, 2013, pp. 595–606.

[15] G. A. Cagri Balkesen, Jens Teubner and M. T. Öszu, "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware," in *ICDE*, 2013, pp. 362–373.

[16] S. Blanas, Y. Li, and J. M. Patel, "Design and evaluation of main memory hash join algorithms for multi-core CPUs," in *SIGMOD*, 2011, pp. 37–48.

[17] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," in *OOPSLA*, 2005, pp. 519–538.

[18] C. Maynard, "Comparing UPC and one-sided MPI: A distributed hash table for gap," in *PGAS*, 2011.

[19] J. M. Malard and R. D. Stewart, "Distributed dynamic hash tables using IBM LAPI," in *SC*, 2002, pp. 1–11.

[20] B. Roussev and J. Wu, "Distributed computing using Java: A comparison of two server designs," *J. Syst. Archit.*, vol. 52, no. 7, pp. 432–440, Jul. 2006.

[21] T. El-Ghazawi and F. Cantonnet, "UPC performance and potential: A npb experimental study," in *SC*, 2002, pp. 1–26.

[22] A. R. Karlin and E. Upfal, "Parallel hashing: An efficient implementation of shared memory," in *Journal of the ACM*, vol. 35, no. 4, pp. 876–892, 1988.

[23] Y. Matias and U. Vishkin, "On parallel hashing and integer sorting," *Journal of Algorithms*, vol. 12, no. 4, pp. 573–606, 1991.

[24] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Real-time parallel hashing on the GPU," *ACM Trans. Graph.*, vol. 28, no. 5, pp. 154:1–154:9, Dec. 2009.

[25] I. García, S. Lefebvre, S. Hornus, and A. Lasram, "Coherent parallel hashing," *ACM Trans. Graph.*, vol. 30, no. 6, pp. 161:1–161:8, Dec. 2011.

[26] M.-C. Albutiu, A. Kemper, and T. Neumann, "Massively parallel sort-merge joins in main memory multi-core database systems," *Proc. VLDB Endow.*, vol. 5, no. 10, pp. 1064–1075, Jun. 2012.