A Top-Down Parallel Semisort

Yan Gu
Carnegie Mellon University
yan.gu@cs.cmu.edu

Julian Shun
Carnegie Mellon University
jshun@cs.cmu.edu

Yihan Sun
Carnegie Mellon University
yihans@cs.cmu.edu

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

ABSTRACT

Semisorting is the problem of reordering an input array of keys such that equal keys are contiguous but different keys are not necessarily in sorted order. Semisorting is important for collecting equal values and is widely used in practice. For example, it is the core of the MapReduce paradigm, is a key component of the database join operation, and has many other applications.

We describe a (randomized) parallel algorithm for the problem that is theoretically efficient (linear work and logarithmic depth), but is designed to be more practically efficient than previous algorithms. We use ideas from the parallel integer sorting algorithm of Rajasekaran and Reif, but instead of processing bits of a integers in a reduced range in a bottom-up fashion, we process the hashed values of keys directly top-down. We implement the algorithm and experimentally show on a variety of input distributions that it outperforms a similarly-optimized radix sort on a modern 40-core machine with hyper-threading by about a factor of 1.7–1.9, and achieves a parallel speedup of up to 38x. We discuss the various optimizations used in our implementation and present an extensive experimental analysis of its performance.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sorting and searching*

Keywords

Parallel Algorithms, Semisorting, Integer Sorting

1. INTRODUCTION

The semisorting problem [22] is defined to take as input an array of records with associated keys, and return a reordered array such that records with identical keys are contiguous. Unlike sorting, it does not require that records with distinct keys are ordered. The problem has many applications. In the popular MapReduce paradigm [9], for example, the most expensive step is typically the so-called shuffle step, which collects the tuples with equal keys returned from the map stage together so the reducer can be applied to each group. Also in the relational join operation common in database processing [6], equal values of a field of a relation have to be put together with equal values of a field of another. Indeed in practice, the most recent work on analyzing the performance of in-memory database joins has focused on hash and sorting based methods for semisorting [1]. Most database languages also have a direct `groupBy` operation that groups together records by a given key.

Theoretically, semisorting also has many applications. The term was coined by Valiant in developing techniques for simulating various parallel machine models with other machine models [22]. In such simulations, memory operations to the same location are collected so they can be combined. Semisorting can also be used in many divide-and-conquer algorithms to collect together the parts that go to each recursive call [18], and to collect values associated with vertices in a graph [12]. Bast and Hagerup cite many other applications of semisorting [2].

As with sorting, there are several variants on semisorting depending on the type of keys and the operations allowed. The semisorting problem we consider here is defined as follows: given an array of n records each containing a key from a universe U, along with a family of hash functions $h : U \rightarrow [1, \ldots, n^k]$, for some constant k, and an equality test $f : U \times U \rightarrow \mathit{Boolean}$, return an array of the same records such that the only records between two equal records are other equal records. Other authors have considered semisorting applied to a bounded set of integer keys in the range $[1, \ldots, n]$ [2, 18]. From a theoretical perspective, this is equivalent to our definition since a hash table can be used to first assign unique labels in the range $[1, \ldots, n]$ to each key (assuming the hash table is asymptotically as fast as the semisort). Here we consider the more general definition since it allows for more practical implementations than first assigning unique labels and then performing integer sorting.

Sequential semisorting can be performed by maintaining a hash table in which each entry is a list of records with equal valued keys. The records can then be inserted one at a time. Whenever a key is first encountered a new list is added to the table, and when a key is encountered again, the record is added to the existing list for that key. This simple randomized technique takes linear expected work. Semisorting can also be implemented in linear work by hashing into range

SPAA '15, June 13 - 15, 2015, Portland, OR, USA.
Copyright 2015 ACM ISBN 978-1-4503-3588-1/15/06 $15.00.
DOI: http://dx.doi.org/10.1145/2755573.2755597.

$[1, \ldots, n^k]$ and then sorting the keys using an integer sort. For sufficiently large k (i.e. $k > 2$), it is unlikely there are collisions.

Parallel semisorting is not as simple. Although parallel hash tables that insert each distinct key once are simple and efficient in parallel [19], what makes semisorting hard is the need to insert all the duplicate entries, which need to be collected together. Given that there are many efficient parallel sorting algorithms, one might instead consider an approach based on integer sorting. Interestingly, however, it is not known how to sort integer keys in the range $[1, \ldots, n^k]$, $k > 1$ in linear work and polylogarithmic depth.[1] This problem has been open for almost 30 years with many researchers working on it. What is known is how to sort (but not stably) integers in the range $[1, \ldots, n \log^k n]$ in $O(kn)$ work and $O(k \log n)$ depth [16] with high probability.[2] As mentioned above, this can be used to solve the semisorting problem after a preprocessing step that reduces the integer range. In practice, however, this is not a competitive approach since just the initial preprocessing using a hash table requires about as much work as the whole sequential algorithm that simply inserts the keys into a hash table. Furthermore the integer sort is itself quite complicated.

In this paper, our goal is to develop a theoretically efficient parallel semisorting algorithm (linear work and polylogarithmic depth) that also performs well in practice, ideally closely matching the work done by the sequential algorithm. Our algorithm uses many of the ideas from the Rajasekaran and Reif integer sort [16] but instead of working bottom-up (i.e. least-significant bits first) on the bits generated from a reduced integer range, it works top-down (most-significant bits first) directly on the hash values. It significantly improves on constant factors while matching the theoretical bounds.

Our algorithm works as follows. We first hash the keys of the records into enough bits so that collisions are unlikely. We then take a sample of these hashed keys and sort them. The sorted hashes are used to predict the number of keys in each range of the hash, as well as to detect keys with many duplicates, which we refer to as *heavy keys*. The remaining keys are referred to as *light keys*. Next, appropriately-sized arrays are allocated for each of the heavy keys and for ranges of the light keys. Then, the records are all written to random locations within their appropriate array, retrying at the next location when there is a collision. Finally, a counting sort is used on each array of records associated with light keys, which are reasonably small. Each step takes at most linear work and logarithmic depth.

In practice the algorithm is quite efficient since much of the work is in the single write to memory for each key (when writing to the allocated arrays). The other memory operations performed by the algorithm are computations on a smaller sample (the sample sort), reads from a much smaller sample (determining the array for writing), adjacent writes typically on the same cache line (when colliding on a write, and using an optimization that we describe), linear scans of the data (to read the records, and to compact the arrays), or reads and writes within a polylogarithmic-sized block of keys (for the final sorts).

We implement algorithm and describe the results of a variety of experiments to measure its performance and compare it to other methods. Our experiments are performed on a 40-core machine with two-way hyper-threading. On all 40 cores (with hyper-threading enabled), our code achieves up to a factor of 38 speedup over its performance on a single thread. Also even on one thread, our algorithm is 20% faster than a simple sequential version using a hash table. This is because the sequential version requires using linked lists to link the elements going to the same bucket, which is not as efficient as estimating sizes and writing directly to an array.

We measure our algorithm on a variety of different distributions and show that it is robust across the distributions, only differing in running time by about 20% between the best and worst distribution. Our experiments are done on a variety of sizes ranging from 10 million records to one billion records. The per-record performance improves for larger inputs. We also present a breakdown of the running time into the components of the algorithm, and show that the time is dominated by the single loop that writes the records into their respective arrays. In addition, we compare our algorithm to a radix sort from an existing library and show that it is about twice as fast. We finally compare our algorithm to comparison sorting algorithms from existing libraries, and our results show that our algorithm outperforms them on large inputs.

2. PRELIMINARIES

We use the work-depth model [13] allowing for concurrent reads and writes, where ***work*** W is equal to the number of operations required (equivalently, the product of the time and the number of processors) and ***depth (span)*** D is equal to the number of time steps required. The parallelism of an algorithm is therefore W/D. For our algorithms, the same bounds can be obtained on the arbitrary CRCW PRAM model. Using Brent's scheduling theorem [5], we can obtain a running time of $W/P + D$ when using P processors.

We use the notation $[n]$ to indicate the range $[1, \ldots, n]$. We make use of a variety of standard problems as building blocks. The ***prefix-sum (scan)*** problem takes an array of n integers and returns an equal length array in which each element is the sum of the previous elements, as well as the overall sum. The ***packing*** problem takes an array of values and an equal length array of flags, and packs the elements at positions with true flags down into a contiguous output array. It can be implemented in parallel with a prefix sum on the flags (treated as 0s and 1s) followed by a write to the resulting positions. The ***naming*** problem takes a set of n keys with m distinct values, and a hash function h on the keys, and output a unique label for each distinct key with a value in the range $[O(m)]$. The problem can be solved easily with a parallel hash table. The ***placement*** problem [18] (also called the assignment problem [16], or multiple compaction problem [10]) takes an input array A of records each with an integer key in the range $[m]$, and also for each key value i an array B_i of size $n_i \geq \sum_{j=1}^{n} [A_j = i]$.[3] It places each record of A somewhere in the array associated with its key.

[1] This statement assumes the standard word length of $O(\log n)$. Linear work and logarithmic depth can be achieved if a non-standard polynomial word length is assumed [17].

[2] We use "with high probability" (w.h.p.) to mean with probability at least $1 - 1/n^c$ for any constant $c > 0$, and with the constant in the big-O linear in c.

[3] $[\cdot]$ is the Iverson bracket.

All of these problems have simple linear work algorithms with $O(\log n)$ depth (using randomization and high probability bounds for naming and placement). See, for example [13]. Furthermore the algorithms are quite efficient in practice. The problems also have significantly more complicated sublogarithmic depth algorithms, at least for approximate versions [16, 11, 2]. In this paper, however, we are satisfied with the more practical logarithmic depth algorithms. We also use comparison-based sorting, which can be implemented using Cole's mergesort in $O(n \log n)$ work and $O(\log n)$ depth [7] or with slightly more depth with a variety of algorithms.

We say that a sorting algorithm is ***stable*** if the output preserves the relative order among equal keys from the input order, and otherwise we say that the algorithm is ***unstable***.

We review the Rajasekaran and Reif integer sorting algorithm [16]—both because it is relevant to the discussion and also because our algorithm uses some of the ideas. The algorithm consists of two components. The first is a unstable randomized sort for integers in the range $[n/\log^2 n]$ and takes $O(n)$ work and $O(\log n)$ span (w.h.p.). The second is a stable counting sort for integers in the range $[m]$, $m \leq n$ and requires $O(n)$ work and $O(m + \log n)$ span. Using these sorts, integers in the range $[n \log^k n]$ can be sorted in $O(kn)$ work and $O(k \log n)$ span (w.h.p.). In particular, one round of the unstable randomized sort is applied on the $\log(n/\log^2 n)$ low-order bits, followed by $k+2$ rounds of the stable counting sort on integers in the range $[\log n]$ on the high-order bits of the keys. Since the counting sort is stable, it maintains the relative order of the randomized sort on the low-order bits.

The stable counting sort is a simple parallel version of sequential counting sort. It partitions the sequence into n/m blocks each of size m, and works in three phases. In the first phase it counts how many keys of each value are in each block. This can be run in parallel across the blocks and sequentially within each block. Then a prefix sum is used to calculate an offset for each key within each block where the keys will be written. Finally each block goes over its elements again and writes them to their final location. The first and last steps take $O(n)$ work and $O(m)$ span. The middle step takes $O(n)$ work and $O(\log n)$ span. The sort is fully deterministic and gives the stated bounds.

The unstable randomized sort consists of four steps: generating an upper bound on the cardinality of each key, allocating a sufficiently sized array for each key value, writing each key into a random location of its array, and packing the result into a contiguous sorted array. The upper bound on the cardinality for each key is determined by taking a randomly selected sample of size $\Theta(n/\log n)$ and sorting it using a parallel comparison sort, giving a count for each key $c(i)$. Using Chernoff bounds, one can show that with high probability the values $u(i) = c' \max(\log^2 n, c(i) \log n)$ give an upper bound on the cardinality of key i (for some constant c'). Furthermore the estimate ensures that $\sum_{i=1}^{m} u(i) = O(n)$ in expectation. Arrays of size $u(i)$ are then allocated using a prefix sum on the $u(i)$ values, giving an offset for each subarray within an array of length $O(n)$. Now each key is written in parallel into some position in its subarray using an algorithm for the placement problem. The final step does a packing operation to remove the empty spots in the resulting array. All steps take $O(n)$ work and $O(\log n)$ depth (w.h.p.), and follow from the the building blocks listed above.

Algorithm 1 Parallel Semisort

```
Input: An array $A$ with $n$ records each containing a key.
Output: An array $A'$ storing the records of $A$ in semisorted order.
1: Hash each key into the range $[n^k]$ $(k > 2)$
2: Select a sample $S$ of the hashed keys, independently with
   probability $p = \Theta(1/\log n)$.
3: Sort $S$.
4: Partition $S$ into two sets $H$ and $L$, where $H$ contains
   the records with keys that appear at least $\delta = \Theta(\log n)$
   times in $S$ (heavy keys), and $L$ contains the remaining
   records (light keys).
5: Create a hash table $T$ which maps each heavy key to its
   associated array.
6: Heavy keys:
     (a) For each distinct hashed key in $H$ allocate an
         appropriately-sized array for it.
     (b) Insert the records in $A$ associated with heavy keys
         (which can be checked by hash table lookup in $T$)
         into their associated array.
7: Light keys:
     (a) Evenly partition the hash range into $\Theta(n/\log^2 n)$
         buckets, and create an appropriately-sized array for
         each bucket by counting light keys in $S$.
     (b) Insert the records in $A$ associated with light keys
         (which again can be checked by hash table lookup
         in $T$) into a random location in the array of its
         associated bucket.
     (c) Semisort each bucket.
8: Pack all of the arrays into a contiguous output array $A'$.
```

3. OUR ALGORITHM

In this section, we describe and analyze our algorithm for semisorting. In the next section, we will describe various implementation decisions we made to improve the performance. The input to our algorithm is an array of n records. Each record contains a key. We assume a uniform random hash function that maps keys to integers in the range $[n^k]$ in constant time. The outline of our parallel semisorting algorithm is given in Algorithm 1. We prove the following theorem regarding the complexity of our algorithm.

THEOREM 3.1. *Algorithm 1 for parallel semisorting can be implemented in $O(n)$ expected work and space, and $O(\log n)$ depth w.h.p.*

The algorithm first hashes the keys into a sufficient large range $[n^k]$ and $k > 2$ so that collisions are unlikely (i.e. there is a one-to-one correspondence between keys and hashed keys). The remainder of the algorithm semisorts these *hashed keys*. Step 2 generates a sample S of the hashed keys, where each hashed key is included in S with probability $p = \Theta(1/\log n)$. This can be done with a parallel pack operation in $O(n)$ work and $O(\log n)$ depth. The expected size of S is $\Theta(n/\log n)$. Step 3 sorts S using Cole's parallel mergesort [7] in $O(n)$ expected work and $O(\log n)$ depth. With S in sorted order we can now compute the multiplicity of each sampled hashed key in S.

We define a hashed key to be a ***heavy key*** if records with that key appear at least $\delta = \Theta(\log n)$ times in the sample S, and a ***light key*** otherwise. A ***heavy record*** is a record associated with a heavy key, and a ***light record*** is a record associated with a light key. Heavy records appear many times in the input, and are handled differently than light records for both theoretical and practical reasons. Step 4 partitions S into heavy keys H and light keys L. This can be done with prefix sums in linear work and logarithmic depth.

Steps 5 and 6 collect all heavy records with the same hashed key into its own bucket. In Step 5, we create a hash table T which maps the hash value of each heavy key to an appropriate array. The arrays for the heavy keys are allocated in Step 6a. For a heavy key appearing s times in S (we know s from previous steps), it will appear $O(s/p)$ times in A w.h.p. In Section 3.1, we provide a precise high probability upper bound on the number of times a key will appear in A, which we denote by $f(s)$. The step allocates an array of size $\alpha f(s)$ ($\alpha > 1$) for each heavy key in the sample. The expected work and space for allocating the arrays is $O(n)$, as we show in Section 3.1 that the sum of all $f(s)$'s is $O(n)$ in expectation. T can be created and filled in $O(n/\log n)$ work and $O(\log n)$ depth w.h.p., and supports $O(1)$ work lookups [11].

Step 6b uses the placement problem (described in Section 2) to place the heavy records into their appropriate array. The placement problem can be implemented by partitioning the input into blocks of size $\log n$ and inserting records in rounds. In each round, we take an uninserted record from each block in parallel, select a random location in its associated array, check if the location is empty, and if so write the record into the location. Each such record then checks to see if it was successfully written (since another block could have also written to the location). If unsuccessful it will continue to the next round, otherwise we move to the next record in the block. Each record has at least a $1 - 1/\alpha$ probability of succeeding in each round since each array is at least α times the size of the number of records destined for it. This means the expected number of rounds is $(\alpha/(\alpha-1))\log n$. Since the rounds are independent, after $O(\log n)$ rounds w.h.p. all blocks finish (using Chernoff bounds). Each round has constant depth. The total work is $(n/\log n) \times O(\log n) = O(n)$.

Step 7 collects all light keys within ranges of the hash space into their own bucket. First, we evenly partition the hash range into $\Theta(n/\log^2 n)$ buckets, and create appropriately-sized arrays for each bucket (Step 7a). This is done by counting the number of records with light keys in S that fall into each bucket using a prefix sum. If s records fall into a bucket, we allocate an array of size $\alpha f(s)$ for the bucket. We can again use the placement problem to insert the records into their appropriate arrays (Step 7b). Unlike the heavy keys, each bucket can contain records with different hashed keys. We therefore now need to semisort within the buckets (Step 7c). To do this work-efficiently we note that w.h.p. there are at most $O(\log^2 n)$ distinct keys per bucket—we are throwing at most n balls (distinct keys) into $\Theta(n/\log^2 n)$ buckets assuming the hash function is uniform [8]. For each bucket, we use the naming problem to give new labels to the keys in the range $[O(\log^2 n)]$ by inserting and then looking up in a hash table. This has expected linear work and $O(\log n)$ depth w.h.p. For each bucket we then use two passes of the stable counting sort on the newly labeled records, each pass sorting $O(\log\log n)$ bits. The work and space summed over all the light key buckets is $O(n)$ and the depth is $O(\log n)$.

Finally, Step 8 uses a parallel pack over all the arrays. Since the expected total size of the arrays is $O(n)$ (see Section 3.1), the expected work and space is $O(n)$, and depth is $O(\log n)$.

Combining the complexity of each step gives Theorem 3.1.

The algorithm's correctness (assuming no collisions in the initial hashing) is easily verified by noting that each heavy key array contains records with the same key, and the light key arrays are sorted so records with the same key appear next to each other. Concatenating the arrays together gives a semisorted output. Because of the initial hashing, this is a Monte Carlo algorithm, but can be converted into a Las Vegas algorithm by checking for correctness of the initial hashing with a parallel hash table and restarting the algorithm if there are collisions. Lastly, although unlikely to happen (Corollary 3.4), it is possible that a bucket can overflow. This can be checked by keeping a counter of the number of trials, and if it exceeds a threshold, the algorithm can be restarted.

3.1 Lemmas for size estimation

Consider some set of keys K. We are interested in estimating the number of records in A that have a key in K, given that there are s such records in S. We define the following function:

$$f(s) = \left(s + c\ln n + \sqrt{c^2\ln^2 n + 2sc\ln n}\right)/p$$

where $p = \Theta(1/\log n)$ is the sampling probability and c is a constant. We will prove that the function f gives a high probability upper bound on the number of records with a key in K in the input array A, given that there are s such records in the sample S. Our motivation for providing a precise upper bound is because in our implementation we must obtain accurate estimates for efficiency.

LEMMA 3.2. *For a sample S in which an element in A is included with probability p, if there are s records in S with a key in K, then the probability that the number of such records in A is greater than $f(s)$ is at most n^{-c}.*

To prove Lemma 3.2, we first prove a lemma stating that for records with a key in K appearing $f(s)$ times in A, the probability that there are at most s such records in S is at most n^{-c}. This can be shown by applying a Chernoff bound with the mean of s being $pf(s)$ in the following lemma. Let σ be the number of such records in S, and ν be the number of such records in A.

LEMMA 3.3. $\Pr[\sigma \le s \,|\, \nu = \lceil f(s) \rceil] \le n^{-c}$.

PROOF.

$$\begin{aligned}
&\Pr[\sigma \le s \,|\, \nu = \lceil f(s) \rceil] \\
&\le \exp\left[-\left(1 - \frac{s}{pf(s)}\right)^2 \cdot pf(s)/2\right] \\
&= \exp\left[s - \frac{1}{2}\cdot\left(pf(s) + \frac{s^2}{pf(s)}\right)\right] \\
&= \exp\left[s - \frac{1}{2}\left(pf(s) + \frac{s^2\left(s + c\ln n - \sqrt{\Delta}\right)}{(s + c\ln n)^2 - \Delta}\right)\right] (*) \\
&= \exp\left[s - \frac{1}{2}(2s + 2c\ln n)\right] \\
&= \exp[-c\ln n] = n^{-c}
\end{aligned}$$

where $\Delta = \big(c^2 \ln^2 n + 2sc \ln n\big)$ on the line marked $(*)$. □

With Lemma 3.3, we now prove Lemma 3.2.

Proof of Lemma 3.2. Applying the law of total probability, we have:

$$\begin{aligned}
&\Pr[f(\sigma) \le \nu] \\
&= \sum_{\nu'} \Pr\big[f(\sigma) \le \nu' \mid \nu = \nu'\big] \Pr\big[\nu = \nu'\big] \qquad (1)
\end{aligned}$$

We first rearrange the terms so that we can apply Lemma 3.3. We use the fact that $f(s)$ is a monotonically increasing function.

$$\begin{aligned}
&\Pr\big[f(\sigma) \le \nu' \mid \nu = \nu'\big] \\
&= \Pr\big[f(\sigma) \le f\big(f^{-1}(\nu')\big) \mid \nu = f\big(f^{-1}(\nu')\big)\big] \\
&= \Pr\big[\sigma \le f^{-1}(\nu') \mid \nu = f\big(f^{-1}(\nu')\big)\big] \qquad (2)\\
&\le n^{-c} \qquad (3)
\end{aligned}$$

We obtain (3) from (2) by plugging in $s = f^{-1}(\nu')$ into Lemma 3.3. Plugging this into (1), we have

$$\begin{aligned}
&\Pr[f(\sigma) \le \nu] \\
&= \sum_{\nu'} \Pr\big[f(\sigma) \le \nu' \mid \nu = \nu'\big] \Pr\big[\nu = \nu'\big] \\
&\le \sum_{\nu'} n^{-c} \Pr\big[\nu = \nu'\big] \\
&= n^{-c} \sum_{\nu'} \Pr\big[\nu = \nu'\big] \\
&= n^{-c}
\end{aligned}$$

which gives the high probability bound. □

We now apply Lemma 3.2 for each bucket in Algorithm 1. For heavy key buckets K is a single key, whereas for light key buckets K is the set of keys falling in the range of the bucket. Over all buckets, $f(s)$ is an upper bound on the number of records appearing in the bucket with probability at least $1 - \Theta(n^{-c+1}/\log^2 n)$ by applying a union bound with Lemma 3.2 (there are at most $\Theta(n/\log^2 n)$ buckets). This gives us the following corollary:

Corollary 3.4. *The probability that f gives an upper bound on the number of records in each bucket is at least $1 - \Theta(n^{-c+1}/\log^2 n)$.*

Picking $c > 1$ gives a high probability bound as required in the proof of Theorem 3.1. The constant hidden by $\Theta(\cdot)$ can be arbitrarily small since it is decided by the default constants p, δ, and the number of buckets for light keys.

We will now prove that using the function f, the sum of estimates over all buckets is $O(n)$ in expectation. Recall that the n input records are partitioned into $\Theta(n/\log^2 n)$ buckets for the light keys and $O(n/\log^2 n)$ buckets (expected) for the heavy keys (the sample size is $\Theta(n/\log n)$ and heavy keys appear $\delta = \Omega(\log n)$ times in the sample). Therefore there are a total of $\Theta(n/\log^2 n)$ buckets in expectation. Let R denote the number of buckets, and let s_i denote the number of times records belonging to bucket i appear in the sample S.

Lemma 3.5. *$\sum_{i=1}^{R} f(s_i) = \Theta(n)$ holds in expectation.*

Proof. Note that $E\Big[\sum_{i=1}^{R} s_i\Big] = E[|S|] = \Theta(n/\log n)$. Therefore,

$$\begin{aligned}
E\left[\sum_{i=1}^{R} f(s_i)\right] &= \sum_{i=1}^{R} E[f(s_i)] \\
&= \sum_{i=1}^{R} E\Big[\Big(s_i + c\ln n + \sqrt{c^2 \ln^2 n + 2 s_i c \ln n}\Big)/p\Big] \\
&\le \frac{1}{p}\sum_{i=1}^{R} E[s_i] + \frac{2c}{p} R \ln n + \frac{1}{p}\sum_{i=1}^{R} E\Big[\sqrt{2 s_i c \ln n}\Big] \\
&\le \Theta(\log n)\cdot\Theta(n/\log n) \\
&\quad + 2c\cdot\Theta(\log n)\cdot\Theta(n/\log^2 n)\ln n \\
&\quad + \frac{1}{p}\cdot\sqrt{2c\ln n}\cdot E\left[\sqrt{R\sum_{i=1}^{R} s_i}\,\right] \qquad (4)\\
&= \Theta(n) + \Theta(n) + \sqrt{2c\ln n}\cdot \ln n\cdot \Theta\left(\sqrt{\frac{n^2}{\log^3 n}}\right) \\
&= \Theta(n)
\end{aligned}$$

where we apply the Cauchy-Schwarz inequality at (4). □

Since we allocate arrays of size $\alpha f(s)$ for each bucket, Lemma 3.5 implies that the total work and space required for allocating and packing the the arrays is linear in expectation, as required by Theorem 3.1.

3.2 Comparison to integer sorting

Our semisorting algorithm uses various ideas from Rajasekaran and Reif's (RR) integer sorting algorithm [16]. Also as mentioned the RR algorithm can be used for the semisorting problem by first using the naming problem (with a hash table) to reduce the range of the hash values to $[n]$ and then integer sorting (recall that RR is limited to keys in the range $[n \log^k n]$). Here we discuss the differences between the two approaches and why we made the various choices, with the view of developing a more practical algorithm in mind. Firstly, we do not need to reduce the range of the hash values and instead can work directly with hashed keys in the range $[n^k]$, with k picked so collisions are unlikely. This avoids an extra hashing step across all keys. This is possible since the hashed keys are uniformly distributed. This played a critical role in bounding the number of distinct keys in each light bucket.

Secondly, we separate light keys from heavy keys. This has the advantage that the heavy records can immediately be placed in their correct bucket without the need for a secondary sort. This is important in practice because there can be many (perhaps all) equal valued keys meaning that buckets with heavy keys can be very large making sorting more expensive. The remaining light buckets have at most $O(\log^3 n)$ keys (w.h.p.) and have expected size $O(\log^2 n)$. Therefore sorting them is very cache-friendly. Also, from a practical standpoint, the hash table of heavy keys can be made small enough to fit in cache, as can the array of pointers to the light buckets.

Thirdly, the RR algorithm for keys in the range $[n]$ uses two rounds of the stable counting sort to start with after applying the randomized unstable sort. These are expensive since they create global movement. Instead our algorithm applies the stable counting sort separately on buckets that are always small (polylogarithmic size). More discussion of practical aspects of our particular implementation are described in the next section.

4. IMPLEMENTATION DETAILS

In this section, we discuss some of the details of our implementation. To start with, we introduce the default constants in the algorithm. We set the sampling probability p to be 1/16, and δ to be 16, which we found to give the best overall performance in our experiments for our range of input sizes (10^7 to 10^9 records). The number of light key buckets is set to be 2^{16}. Our implementation heavily based on the Problem Based Benchmark Suite (PBBS) [20] which contains simple and efficient parallel code to a number of problems and parallel primitives, including prefix sum, filter/pack, radix sort, and concurrent hash tables based on linear probing [19].

The implementation of this algorithm is broken down into five phases, and this is also the breakdown that we use in the detailed experimental analysis in Section 5.

Phase 1: Sampling and sorting. This phase corresponds to Steps 2 and 3 in Algorithm 1. When sampling, the i'th sample is randomly picked from the $(\lceil(i-1)/p\rceil + 1)$'th to the $\lceil i/p\rceil$-th record. Theoretically, for each key, the average number of samples using this sampling scheme is the same as the method that picks every sample independently.

To sort the samples in S, we use the parallel radix sort in the PBBS. The radix sort is a top-down sort, which processes 8 bits of the key at a time to place the records into buckets, and recurses on each bucket. This parallel radix sort is also the baseline algorithm we compare against in Section 5.

Phase 2: Bucket allocation. In this phase we perform Steps 4, 5, 6a and 7a in Algorithm 1. Since this phase is inexpensive relative to the whole algorithm (about 1% of the overall running time), we use a straightforward implementation. To filter out heavy keys and get their counts, we first compute the offsets corresponding to the start of each key in the sorted array, which can be done with a simple comparison with the preceding key. We then gather these offsets using a parallel filter, and finally compute the counts by using the difference between consecutive offsets. If the count for a key is greater than $\delta = 16$, we insert the key into a hash table. This hash table stores pointers to the arrays associated with heavy keys, and the arrays are allocated using the size computed from the f function in Section 3.1. All of these steps run in parallel.

We also use the f function to compute the size of the light key arrays, but use a minor optimization where we combine small (adjacent) buckets into a single bucket corresponding to at least δ light key records in S. This optimization reduces the overall running time by at most 10%. This is because the estimation function f is more accurate with a larger number of samples, and therefore the overall used memory space is reduced. Then the following steps (local sort and packing phases) will touch less memory, and this improves the running time.

We sequentially create the associated arrays for the heavy key and light key buckets since it is a very small fraction of the running time. To allow for efficient packing later, we use a single large array for all of the buckets, and each bucket simply stores an offset into this array to indicate the start of its associated array. The heavy key buckets are all before the light key buckets in the array. Each bucket with s samples allocates an array of size $1.1f(s)$ with $c = 1.25$, and rounded up to the nearest power of 2. In our experiments, this size was sufficient to prevent overflow on all of our inputs.

Phase 3: Scattering. This phase corresponds to Steps 6b and 7b in Algorithm 1, where every record is scattered to a random location in the array of its bucket. In contrast to the discussion in Section 3, we perform the insertions using a compare-and-swap, which is supported on most modern multicore machines. A compare-and-swap returns *true* if a record was successfully inserted into an initially empty location and *false* otherwise. On a failure, instead of picking another random location, a record tries the next location (linear probing). This gives better cache performance. Bounds for linear probing show that the expected cost per insertion is $O(1)$, leading to linear work. Furthermore, the largest cluster in the array is $O(\log n)$ w.h.p., which gives a depth bound of $O(\log n)$ w.h.p. [8].

Phase 4: Local sort. This phase corresponds to Step 7c in Algorithm 1. After all the records are inserted into the buckets, a pack followed by a local sort is executed on each bucket. In our implementation, the local sort in each array is sequential since sorting a single array is fast, and usually there are many more arrays than processors, so this step has good parallelism. We tried several versions including a bucket sort, some comparison-based hybrid sort algorithms, and the sort in the C++ Standard Library (STL). The running times for the various algorithms were similar. In our final implementation, we choose to use the sort in the C++ Standard Library, which is implemented using a hybrid of quicksort, heap sort and insertion sort, since it provided consistent performance on all of our input distributions.

Phase 5: Packing. This phase corresponds to Step 8 in Algorithm 1. The algorithm that we use to pack the portion of the array for the heavy key buckets consists of 3 steps (recall that we use a single array A' to represent all the buckets): first, the array A' is divided into 1000 intervals and each interval is packed individually and sequentially by just scanning the interval; second, we apply a sequential prefix sum on the counts for the intervals to compute the boundaries in A' for each interval; finally, we write the records into their appropriate indices in A' in parallel. The portion of the array for the light key buckets is already packed from Phase 4 so we simply copy the records into A' in parallel.

5. EXPERIMENTS

We measure the performance of the implementation of our parallel semisorting algorithm using various parameters. We also compare the performance of our semisorting algorithm to the parallel integer sorting algorithm from the Problem Based Benchmark Suite (PBBS) [20], which can be used to perform semisorting. Finally, we compare with a sequential implementation of semisorting.

We run our experiments on a 40-core (with two-way hyper-threading) machine with 4 × 2.4GHz Intel 10-core E7-8870 Xeon processors (with a 1066MHz bus and 30MB L3 cache) and 256GB of main memory. We run all parallel experiments with hyper-threading enabled, for a total of 80 hyper-threads. We compile our code with `g++` version 4.8.0 with the `-O2` flag. The parallel codes use Cilk Plus [14] to express parallelism, which is supported by the `g++` compiler that we use. In particular, the parallel for-loops are written using the `cilk_for` construct. Divide-and-conquer parallelism, which is required by the parallel integer sort, is written using the `cilk_spawn` construct. For an algorithm with work W and depth D, Cilk's randomized work-stealing scheduler [4] with P available threads gives an expected running time of $W/P + O(D)$. When running in parallel, we use the command `numactl -i all` to evenly distribute the allocated memory among the processors.

5.1 Input data

All of our experiments use an 8-byte (64-bit) hash value along with 8-byte payload (16 bytes total per record). We assume that the keys have been pre-hashed, since the cost for hashing itself would depend on the particular type of value being semisorted and the cost is common across any of the hash-based techniques (including radix sort). Furthermore, the cost for hashing is typically small. With an 8-byte hash the probability of a collision is small, and checking for collisions is easy once the sort is done.

Although the values are hashed, the distribution of duplicates can vary significantly. We use a variety input distributions, including uniform distributions, exponential distributions, and Zipfian distributions. Each class of distribution has a parameter. For uniform distributions, the parameter N indicates the range from which the integers are chosen from. More precisely, each key will be chosen uniformly from the range $[N]$. Hence, a smaller N will create more equal keys. The parameter λ for exponential distributions represents the mean of the distribution, and accordingly, the variance of the distribution is λ^2. The parameter M of Zipfian distributions denotes the range $[M]$ that the keys can be chosen from. The i-th number in this range has a probability $1/(i\bar{M})$ of being chosen, where $\bar{M} = \sum_{i=1}^{M} 1/i$ is the normalizing factor.

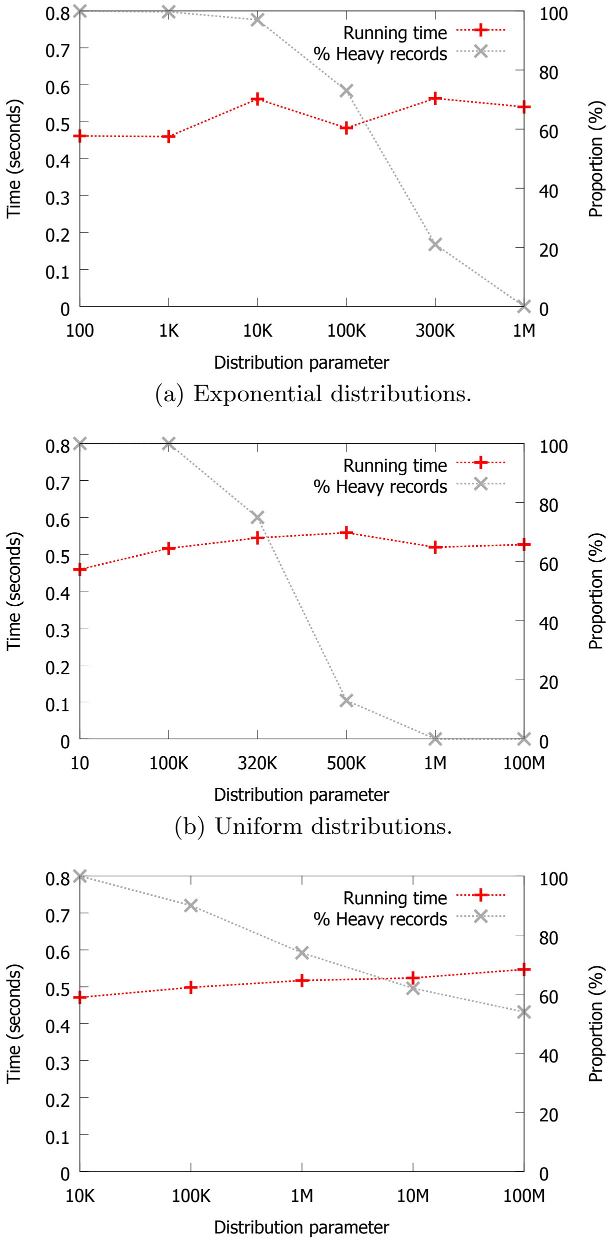
In Section 5.2, we use all three classes of distributions with various parameters to show the stability of our algorithm. In Sections 5.3–5.5 we present a detailed performance analysis using two representative distributions, the uniform distribution with parameter $N = n$ (input size), and the exponential distribution with parameter $\lambda = n/10^3$. These two distributions were chosen because the first one contains only light keys, and the second distribution contains about 30% light keys and 70% heavy keys while not containing too many (as many as a constant fraction of) duplicates. Hence, the performance of both the heavy-key arrays and light-key arrays can be analyzed.

For most of the experiments the input is 100 million 64-bit key-value pairs. However, in Section 5.4 we analyze the performance with various input sizes. All our experiments use 16 byte records in which 8 bytes are the hash of the keys, and the other 8 bytes are the payload.

5.2 Consistency of performance

To show the stability of the parallel semisorting algorithm with different input distributions, we tested the performance on three different classes of distributions with 17 distributions in total. The distributions include different percentages of heavy keys, and span the entire range of 0% to 100%. The average number of duplicates for each key also varies significantly among distributions.

A detailed running time and speedup for each experimental run is reported in Table 1 with different thread counts.



(a) Exponential distributions.

(b) Uniform distributions.

(c) Zipfian distributions.

Figure 1: Running time (seconds) using 40 cores with hyper-threading and the proportion (%) of heavy keys of three different classes of distributions with various parameters. The input size is 10^8.

We also plot the 40-core (with hyper-threading) running time for each class of distributions versus the distribution parameter on 10^8 records in Figure 1.

From the figure, we can see that the lowest running time (0.46s) appears in three test cases, and in all three of these cases, more than 99% records have heavy keys. The algorithm is particularly efficient in this case because no local sort for the light keys is required. Meanwhile, the highest running time is 0.56s, and the common situation in these cases is that most of the keys are close to the threshold between heavy and light key but the majority of keys are

	Sequential		40h		Speedup
	time (s)	%	time (s)	%	
sample and sort	1.03	7.41	0.06	13.29	16.03
construct buckets	0.11	0.77	0.02	3.32	6.65
scatter	9.81	70.60	0.25	51.95	39.08
local sort	0.18	1.30	0.01	1.22	30.51
pack	2.77	19.93	0.15	30.22	18.97

Table 2: Breakdown of running time and percentage for sequential and 40 cores with hyper-threading versions. The input has 10^8 records and the keys follow the exponential distribution with parameter $\lambda = 10^5$. (40h) indicates 40 cores with two-way hyper-threading.

	Sequential		40h		Speedup
	time (s)	%	time (s)	%	
sample and sort	1.55	8.52	0.08	15.18	19.42
construct buckets	0.18	1.00	0.02	3.68	9.40
scatter	9.15	50.25	0.24	45.98	37.81
local sort	6.56	36.02	0.13	23.75	52.48
pack	0.77	4.21	0.06	11.42	12.75

Table 3: Breakdown of running time and percentage for sequential and 40 cores with hyper-threading versions. The input has 10^8 records and the keys follow the uniform distribution with parameter $N = 10^8$. (40h) indicates 40 cores with two-way hyper-threading.

light. Therefore, the size of light key arrays can easily exceed $\Theta(\log^2 n)$, which increases the workload in the local sorting phase.

However, the difference between the extreme cases is only 0.1s, which is about only 20% of the overall running time. This shows that the parallel semisorting algorithm has a reasonably consistent performance on various input distributions.

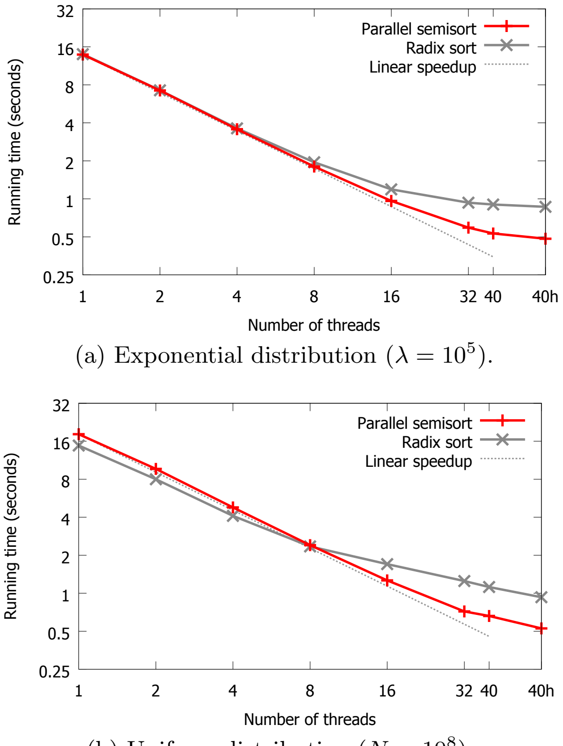
5.3 Performance, speedup, and breakdown

The parallel speedup of the semisorting algorithm is shown in Table 1. The speedup lines for the two representative distributions are also shown in Figure 2 to demonstrate that the algorithm has good parallel speedup. For example, an average speedup of 14.3x is obtained using 16 threads. Moreover, the speedups for 40 cores with hyper-threading are 31.7 and 34.6 on the two input distributions, respectively. Given that some subroutines in the algorithm are memory-bandwidth bound, this speedup is quite good.

To further analyze the performance of the algorithm, we show the breakdown of running time among the different phases in Tables 2 and 3, and Figure 3. The array construction is inexpensive, and hence a lower speedup is tolerable. The speedup of the sampling/sorting phase is determined by the radix sort and is between 16 and 20. The packing phase is memory-bandwidth bound, and gets 12–19x speedup. The scatter process achieves 37–39x speedup. The highest speedup comes from the local sort (30–52x). Since all the light-key arrays fit into caches, this phase is dominated by the computation and not the memory access. The overall cost is dominated by the scatter.

5.4 Scalability with varying input sizes

In this section we analyze the performance of our algorithm as a function of input size. We use inputs with 7 different sizes from 10^7 to 10^9. The running times and speedups for the two representative distributions for these sizes are shown in Table 4. The speedups for the algorithm using 40 cores with hyper-threading are also plotted in Figure 4(a)–(b) and the corresponding speed (records per second) is shown in Figure 4(c)–(d). From the figure we can see that the speedup improves from 23 to 35.2 for the exponential distribution with increasing input size. Similarly for the uniform distribution the speedup improves from 25.2 to 38.2. The speedup for both distributions is high, and higher



(a) Exponential distribution ($\lambda = 10^5$).

(b) Uniform distribution ($N = 10^8$).

Figure 2: Running times (seconds) of parallel semisort and radix sort with varying number of threads on an input size of 10^8. (40h) corresponds to 40-cores with two-way hyper-threading. The linear speedup line is plotted for reference.

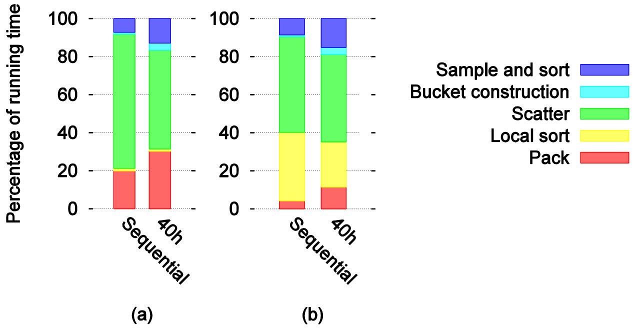
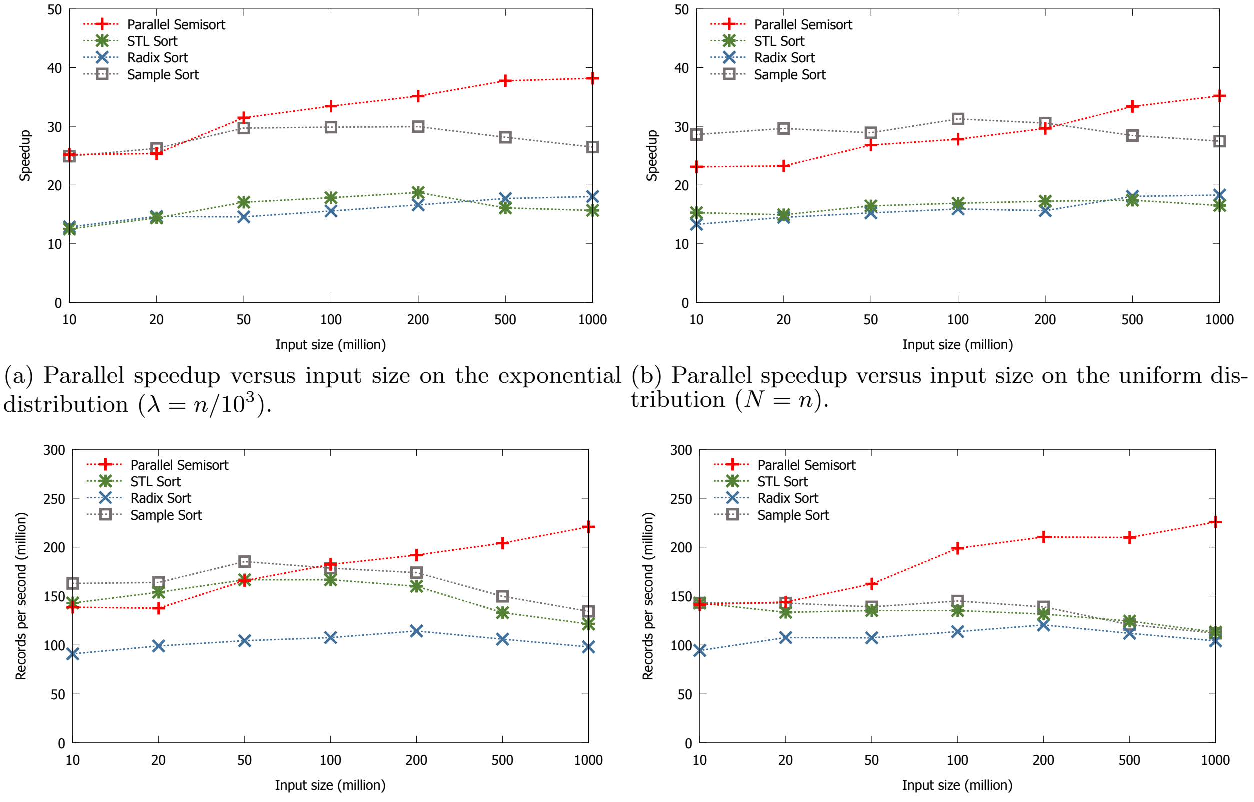


Figure 3: Breakdown of the different phases in terms of percentage of the total running time for running sequentially and on 40 cores with hyper-threading (40h). The input has 10^8 records and the keys follow: (a) exponential distribution with parameter $\lambda = 10^5$, (b) uniform distribution with parameter $N = 10^8$. The data for this figure is taken from Table 2 and 3.

	Indicator	#threads	Exponential distribution						Uniform distribution						Zipfian distribution				
Parameter			100	1K	10K	100K	300K	1M	10	100K	320K	500K	1M	100M	10K	100K	1M	10M	100M
% Heavy key records			99.97	99.7	97	73	21	0	100	100	75	13	0	0	100	90	74	62	54
Parallel semisort	Time	1	13.46	13.48	17.82	13.90	17.24	18.56	12.43	14.73	15.77	18.54	17.50	18.21	13.01	14.51	15.47	16.28	17.83
		2	6.95	7.00	9.34	7.22	8.79	10.00	6.54	7.63	8.10	9.73	9.36	9.67	6.81	7.52	8.22	8.69	9.43
		4	3.46	3.47	4.65	3.56	4.27	4.92	3.26	3.73	3.94	4.82	4.60	4.78	3.39	3.71	4.09	4.26	4.66
		8	1.75	1.76	2.34	1.80	2.17	2.48	1.67	1.89	1.98	2.45	2.33	2.40	1.72	1.88	2.04	2.14	2.38
		16	0.94	0.94	1.25	0.96	1.16	1.30	0.90	1.01	1.06	1.27	1.22	1.26	0.93	1.01	1.09	1.14	1.25
		32	0.57	0.58	0.73	0.59	0.71	0.74	0.56	0.62	0.67	0.79	0.70	0.72	0.57	0.64	0.66	0.69	0.73
		40	0.54	0.53	0.66	0.53	0.65	0.72	0.52	0.59	0.60	0.67	0.64	0.66	0.54	0.56	0.61	0.62	0.66
		40h	0.46	0.46	0.56	0.48	0.56	0.54	0.46	0.52	0.54	0.56	0.52	0.53	0.47	0.50	0.52	0.52	0.55
	Speedup	2	1.94	1.93	1.91	1.92	1.96	1.86	1.90	1.93	1.95	1.90	1.87	1.88	1.91	1.93	1.88	1.87	1.89
		4	3.89	3.89	3.83	3.90	4.04	3.77	3.81	3.95	4.00	3.84	3.80	3.81	3.84	3.91	3.78	3.82	3.83
		8	7.68	7.67	7.61	7.71	7.94	7.50	7.46	7.79	7.98	7.58	7.52	7.58	7.56	7.73	7.56	7.59	7.49
		16	14.38	14.39	14.23	14.43	14.87	14.25	13.77	14.53	14.85	14.57	14.36	14.41	14.07	14.41	14.18	14.29	14.31
		32	23.55	23.30	24.43	23.50	24.19	25.15	22.24	23.59	23.58	23.54	24.83	25.37	22.68	22.69	23.58	23.57	24.53
		40	25.07	25.64	27.21	26.11	26.45	25.89	23.84	25.01	26.11	27.63	27.55	27.55	24.32	25.98	25.31	26.39	26.87
		40h	29.15	29.31	31.74	28.76	30.62	34.37	27.07	28.54	28.98	33.20	33.71	34.60	27.61	29.11	29.91	31.06	32.57
Radix sort	Time	1	12.00	10.33	14.00	13.70	13.90	14.50	14.10	13.80	13.80	13.80	13.90	14.80	13.10	13.60	13.80	14.20	14.00
		40h	0.88	0.88	0.90	0.88	0.90	0.92	0.91	0.89	0.89	0.90	0.90	0.93	0.93	0.94	0.92	0.96	0.90
	Speedup	40h	13.59	11.79	15.56	15.57	15.43	15.83	15.48	15.45	15.47	15.37	15.51	15.90	14.10	14.55	14.95	14.85	15.50

Table 1: Running times (seconds) and speedup of parallel semisort and radix sort on various distributions using a 40-core machine. (40h) indicates 40 cores with two-way hyper-threading. The input size is 10^8.



(a) Parallel speedup versus input size on the exponential distribution ($\lambda = n/10^3$).

(b) Parallel speedup versus input size on the uniform distribution ($N = n$).

(c) Million records/second processed versus input size on the exponential distribution ($\lambda = n/10^3$).

(d) Million records/second processed versus input size on the uniform distribution ($N = n$).

Figure 4: Parallel speedup and records per second (million) for four different algorithms on two distributions with varying input size from $n = 10^7$ to $n = 10^9$, using 40 cores with hyper-threading.

on the uniform distribution since the local sort has very high parallelism across the different buckets. The lower speedup for smaller input sizes is likely due to overhead of parallelism on smaller data.

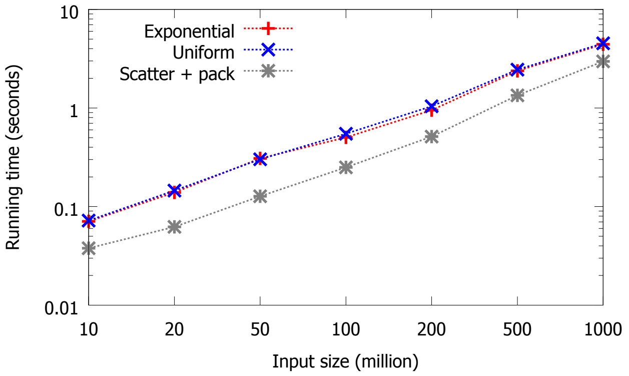


Figure 5: Parallel running times (seconds) with varying input size from $n = 10^7$ to $n = 10^9$, and the comparison to a scatter and pack operation.

As a baseline, we compare the performance of our semisorting algorithm to just a scatter and pack (the minimal work one would need to do to perform semisorting) and show that our algorithm is not much more expensive. In particular, Table 4 and Figure 5 show that on both the uniform and exponential distributions, our semisorting algorithm is just 1.5–2 times slower in parallel than a scatter followed by a pack on an array of size n, with better relative performance as we increase the input size.

We also compared with a simple sequential chained hash table-based algorithm for semisorting and found our algorithm to be 20% faster on a single thread. This is because the sequential implementation requires linked lists to link the elements going to the same bucket, which is not as efficient as estimating sizes and writing directly to an array. In addition, we tried other sequential implementations (e.g. STL vectors, hash tables using open addressing on keys and separate chaining on records with the same key, and a two-phase approach where we simply count the multiplicity of each key, allocate enough space for each key, and write the records into the appropriate locations) but found them to be even less efficient.

5.5 Comparison with other sorting implementations

In this section we compare the performance of our parallel semisorting algorithm with other optimized parallel sorting algorithms: radix sort, STL sort, and sample sort.

We compare the performance of our algorithm to the parallel radix sort in PBBS, which is a top-down recursive approach. The code for the radix sort is equally-optimized and is also a subroutine in our parallel semisorting code. The sequential and parallel running time of the radix sort is provided in Table 1 and Figure 2. The results show that the sequential running times for both algorithms are similar, but the parallel semisort gets about twice the parallel speedup compared to the radix sort. This is because the radix sort works by executing many rounds over the data, thus involving more reads and writes to memory, which limits the speedup as memory bandwidth is the bottleneck.

We also looked at the highly-optimized radix sort of [15], which is the fastest radix sort that we are aware of. Their code is highly-optimized with over 15,000 lines of code. It makes heavy use of AVX vector instructions, which are not supported on our 40-core Intel Nehalem test machine, so we are unable to directly compare with our reported numbers. Based on our experiments on another machine supporting AVX instructions, their code is faster than our code on the uniform random distribution (a particularly easy case), but did not work on more skewed distributions. We believe that their code is designed to only handle well-balanced distributions, and so it is not surprising that it outperforms our code. Furthermore, their code is much more complicated than ours.

In addition, we compare our algorithm with two optimized comparison sorts: GNU libstdc++ (STL) parallel sort [21] implemented with OpenMP and sample sort [3] implemented with Cilk Plus in PBBS [20]. The experiment is run on inputs of varying sizes, ranging from 10^7 to 10^9, and two representative distributions: exponential and uniform distributions. The sequential and parallel running times are shown in Table 5, and the speedup and records per second comparing to parallel semisort are provided in Figure 4.

Although the work complexity of these two algorithms is $O(n \log n)$, they are more cache-friendly, and so their performance is competitive with our parallel semisort. In parallel, the comparison sorts are faster than our semisort on the uniform distribution with an input size of no more than 20 million, and exponential distribution with an input size of no more than 50 million. The STL sort is efficient sequentially (faster than all other algorithms on all inputs), but only has moderate speedup of at most 20 on all cases. The sample sort is designed as a cache-efficient algorithm so it gets consistent speedup of about 30 on all inputs. However, since the work of the comparison sorts is super-linear, their performance (records per second) decreases as the input size grows past 100 million. In contrast, the semisort algorithm scales better since it does linear work.

In our experiments, radix sort is the least efficient in almost all cases. This is because the radix sort is designed for sorting keys from a small range, and the 64-bit keys used in our experiments require too many rounds to sort.

6. CONCLUSION

We have described a parallel algorithm for semisorting that requires linear work and space and logarithmic depth. The algorithm is both theoretically efficient and also practical. We show experimentally that it achieves good parallel speedup on various input distributions and input sizes, and outperforms similarly-optimized comparison and radix sorts for large inputs.

Acknowledgments

This work is partially supported by the National Science Foundation under grant CCF-1218188, and by the Intel Science and Technology Center for Cloud Computing.

7. REFERENCES

[1] C. Balkesen, G. Alonso, J. Teubner, and M. T. Ozsu. Main-memory hash joins on modern processor architectures. In *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2014.

[2] H. Bast and T. Hagerup. Fast parallel space allocation, estimation and integer sorting. *Information and Computation*, 123(1):72–110, 1995.

[3] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low depth cache-oblivious algorithms. In *Proc. ACM*

#records (million)	Exponential distribution				Uniform distribution				40h Running time (sec.)		
	Running time (sec.)		Speedup	Records / sec. (million)	Running time (sec.)		Speedup	Records / sec. (million)	Scatter	Pack	Scatter + Pack
	Sequential	40h			Sequential	40h					
10	1.64	0.07	23.09	141.20	1.82	0.07	25.17	138.56	0.02	0.02	0.04
20	3.23	0.14	23.24	143.92	3.69	0.15	25.36	137.54	0.03	0.03	0.06
50	8.25	0.31	26.80	162.36	9.49	0.30	31.45	165.70	0.07	0.06	0.13
100	13.99	0.50	27.81	198.79	18.34	0.55	33.43	182.29	0.14	0.11	0.25
200	28.18	0.95	29.64	210.33	36.64	1.04	35.14	191.85	0.28	0.23	0.51
500	79.53	2.38	33.38	209.85	92.47	2.45	37.75	204.11	0.70	0.64	1.35
1000	155.90	4.43	35.19	225.69	173.03	4.53	38.19	220.69	1.62	1.35	2.97

Table 4: Running time (seconds), speedup, and records per second (million) for input with varying sizes from 10^7 to 10^9, and the running time for a scatter, a pack and both operations. (40h) indicates 40 cores with two-way hyper-threading.

#records (million)	GNU STL Sort				Sample Sort				Radix Sort			
	Exponential		Uniform		Exponential		Uniform		Exponential		Uniform	
	Seq.	40h	Seq.	40h	Seq.	40h	Seq.	40h	Seq.	40h	Seq.	40h
10	0.88	0.07	1.07	0.07	1.53	0.06	2.00	0.07	1.46	0.11	1.36	0.11
20	1.87	0.13	2.24	0.15	3.20	0.12	4.15	0.14	2.93	0.20	2.72	0.19
50	5.12	0.30	6.08	0.37	8.02	0.27	10.41	0.36	7.30	0.48	6.79	0.47
100	10.70	0.60	12.50	0.74	16.70	0.56	21.56	0.69	14.80	0.93	13.70	0.88
200	23.40	1.25	26.20	1.52	34.40	1.15	44.00	1.44	27.30	1.75	27.60	1.66
500	60.50	3.76	70.00	4.02	93.90	3.34	118.00	4.14	85.30	4.72	79.00	4.47
1000	129.00	8.23	146.00	8.84	197.00	7.44	245.00	8.93	186.00	10.20	173.00	9.60

Table 5: Sequential and parallel running time (seconds) for input with varying sizes from 10^7 to 10^9. (40h) indicates 40 cores with two-way hyper-threading.

Symp. on Parallelism in Algorithms and Architectures (SPAA), pages 189–199, 2010.

[4] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM (JACM)*, 46(5), Sept. 1999.

[5] R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM (JACM)*, 21(2):201–206, 1974.

[6] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM (CACM)*, 13(6):377–387, June 1970.

[7] R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, 1988.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM (CACM)*, 51(1):107–113, Jan. 2008.

[10] P. B. Gibbons, Y. Matias, and V. Ramachandran. Efficient low-contention parallel algorithms. *Journal of Computer and System Sciences*, 53(3):417–442, 1996.

[11] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *Foundations of Computer Science (FOCS)*, pages 698–710, 1991.

[12] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson. Ordering heuristics for parallel graph coloring. In *Proc. ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 166–177, 2014.

[13] J. Jaja. *Introduction to Parallel Algorithms.* Addison-Wesley Professional, 1992.

[14] C. E. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[15] O. Polychroniou and K. A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 755–766, 2014.

[16] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18(3):594–607, 1989.

[17] S. Rajasekaran and S. Sen. On parallel integer sorting. *Acta Informatica*, 29(1):1–15, 1992.

[18] J. H. Reif and S. Sen. Parallel computational geometry: An approach using randomization. In J. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, chapter 18, pages 765–828. 1999.

[19] J. Shun and G. E. Blelloch. Phase-concurrent hash tables for determinism. In *Proc. ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 96–107, 2014.

[20] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: the Problem Based Benchmark Suite. In *Proc. ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 68–70, 2012.

[21] J. Singler, P. Sanders, and F. Putze. Mcstl: The multi-core standard template library. In *Euro-Par*, pages 682–694. 2007.

[22] L. G. Valiant. Handbook of theoretical computer science (vol. a). chapter General Purpose Parallel Architectures, pages 943–973. MIT Press, 1990.