Distributed Perfect Hashing for Very Large Key Sets (Invited Paper)

Fabiano C. Botelho Daniel Galinkin Wagner Meira Jr. Nivio Ziviani

Department of Computer Science
Federal University of Minas Gerais
Belo Horizonte, Brazil
{fbotelho, dggc, meira, nivio}@dcc.ufmg.br

ABSTRACT

A perfect hash function (PHF) $h : S \rightarrow [0, m-1]$ for a key set $S \subseteq U$ of size n, where $m \geq n$ and U is a key universe, is an injective function that maps the keys of S to unique values. A minimal perfect hash function (MPHF) is a PHF with $m = n$, the smallest possible range. Minimal perfect hash functions are widely used for memory efficient storage and fast retrieval of items from static sets.

In this paper we present a distributed and parallel version of a simple, highly scalable and near-space optimal perfect hashing algorithm for very large key sets, recently presented in [4]. The sequential implementation of the algorithm constructs a MPHF for a set of 1.024 billion URLs of average length 64 bytes collected from the Web in approximately 50 minutes using a commodity PC.

The parallel implementation proposed here presents the following performance using 14 commodity PCs: (i) it constructs a MPHF for the same set of 1.024 billion URLs in approximately 4 minutes; (ii) it constructs a MPHF for a set of 14.336 billion 16-byte random integers in approximately 50 minutes with a performance degradation of 20%; (iii) one version of the parallel algorithm distributes the description of the MPHF among the participating machines and its evaluation is done in a distributed way, faster than the centralized function.

Keywords

Distributed, minimal perfect hash function, large key sets

1. INTRODUCTION

Perfect hashing is a space-efficient way of creating compact representation for a static set S of n keys. Perfect hashing methods can be used to construct data structures storing S and supporting queries to locate a key "$x \in S$" in one probe. For applications with only successful searches, the representation of a key $x \in S$ is simply the value of a perfect hash function $h(x)$, and the key set does not need to be kept in main memory.

A *perfect hash function* (PHF) maps the elements of S to unique values (i.e., there are no collisions), which can be used, for example, to index a hash table. Since no collisions occur, each key can be retrieved from the table with a single probe. A *minimal perfect hash function* (MPHF) produces values that are integers in the range $[0, n-1]$, which is the smallest possible range.

Minimal perfect hash functions are used for memory efficient storage and fast retrieval of items from static sets, such as words in natural languages, reserved words in programming languages or interactive systems, item sets in data mining techniques, routing tables and other network applications, sparse spatial data, graph compression and large web maps representation [1, 6, 7, 8, 10, 11].

The demand to deal in an efficient way with very large key sets is growing. For instance, search engines are nowadays indexing tens of billions of pages and algorithms like PageRank [5], which uses the web graph to derive a measure of popularity for web pages, would benefit from a MPHF to map long URLs to smaller integer numbers that are used as identifiers to web pages, and correspond to the vertex set of the web graph.

The objective of this paper is to present a distributed and parallel version of a perfect hashing algorithm for very large key sets presented in [4]. In this algorithm, the construction of a MPHF or a PHF requires $O(n)$ time and the evaluation of a MPHF or a PHF on a given element of an input S requires constant time. The space necessary to describe the functions takes a constant number of bits per key, depending only on the relation between the size m of the hash table and the size n of the input. For $m = n$ the space usage for the MPHF is in the range $2.62n$ to $3.3n$ bits, depending on the constants involved in the construction and in the evaluation phases. For $m = 1.23n$ the space usage for the PHF is in the range $1.95n$ to $2.7n$ bits. In all cases, this is within a small constant factor from the information theoretical minimum of approximately $1.44n$ bits for MPHFs and $0.89n$ bits for PHFs, something that has not been achieved by previous algorithms, except asymptotically for very large n.

The algorithm presented in [4] is highly scalable. The algorithm increases one order of magnitude in the size of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
INFOSCALE 2008, June 4-6, Vico Equense, Italy
Copyright © 2008 978-963-9799-28-8
DOI 10.4108/ICST.INFOSCALE2008.3518

the greatest key set for which a MPHF was obtained in the literature [2]. This improvement comes from a combination of a novel, theoretically sound perfect hashing scheme that greatly simplifies previous methods, and the fact that it is designed to make good use of the memory hierarchy, using a divide-to-conquer technique. The basic idea is to partition the input key set into small buckets such that each bucket fits in the CPU cache. For this reason this algorithm was called *External Cache-Aware* (ECA) algorithm.

The ECA algorithm presented in [4] allows the generation of PHFs or MPHFs for sets in the order of billions of keys. For instance, if we consider a MPHF that requires 3.3 bits per key to be stored, for 1 billion URLs it would take approximately 400 megabytes. Considering now the time to generate a MPHF, taking the same set of 1.024 billion URLs as input, the algorithm outputs a MPHF in approximately 50 minutes using a commodity PC. It is well known that big search engines are nowadays indexing more than 20 billion URLs. Then, we are talking about approximately 8 gigabytes to store a single MPHF and approximately 1,000 minutes to construct a MPHF. Thus, two problems arise when the input key set size increases: (i) the amount of time to generate a MPHF becomes large for a single machine and (ii) the storage space to describe a MPHF will be unsuitable for a single machine.

In this paper we present a scalable distributed and parallel implementation of the ECA algorithm presented in [4], referred to as *Parallel External Cache-Aware* (PECA) algorithm from now on. The PECA algorithm addresses the two aforementioned problems by distributing both the construction and the description of the resulting functions. For instance, by using a 14-computer cluster the distributed and parallel ECA version generates a MPHF for 1.024 billion URLs in approximately 4 minutes, achieving an almost linear speedup. Also, for 14.336 billion 16-byte random integers evenly distributed among the 14 participating machines the PECA algorithm outputs a MPHF in approximately 50 minutes, resulting in a performance degradation of 20%. To the best of our knowledge there is no previous result in the perfect hashing literature that can be implemented in a distributed and parallel way to obtain better scalability and performance than the results presented hereinafter.

2. NOTATION

In this section we present the notation used throughout this paper.

Definition 1. A *key* is made up by symbols from a finite and ordered alphabet Σ of size $|\Sigma|$.

Definition 2. Let Φ denote the *maximum key length*. Then $L = \Phi \log |\Sigma|$ is the maximum key length in bits[1]. Then we define a *key universe* U of size $u = 2^L$.

Definition 3. Let S be a subset of U containing n keys, where $n \ll u$.

Definition 4. Let $h : U \rightarrow M$ be a *hash function* that maps the keys from U to a given interval of integers $M = [0, m-1] = \{0, 1, \ldots, m-1\}$ (i.e., given a key $x \in U$, the hash function h computes an integer in $[0, m-1]$).

[1]Throughout this paper we denote $\log_2 x$ as $\log x$.

Definition 5. A *perfect hash function* $PHF : S \rightarrow M$ is an injection on $S \subseteq U$ (i.e., for all pair s_1, $s_2 \in S$ such that $s_1 \neq s_2$, then $PHF(s_1) \neq PHF(s_2)$, where $m \geq n$).

Definition 6. A *minimal perfect hash function* $MPHF$: $S \rightarrow M$ is a bijection on $S \subseteq U$ (i.e., each key in S is mapped to a unique integer in M and $m = n$).

To evaluate the performance of the PECA algorithm presented in Section 4 we use two metrics: *speedup* and *scale-up*. By fixing the problem size, the speedup refers to how much a parallel algorithm is faster than a corresponding sequential algorithm and is defined as:

Definition 7. The *speedup* $\mathcal{S}_p$ of a parallel algorithm using p processors is:

$$\mathcal{S}_p = \frac{T_1}{T_p}, \tag{1}$$

where T_1 is the execution time of the sequential algorithm and T_p is the execution time of the parallel algorithm with p processors.

Definition 8. The *efficiency* $\mathcal{E}_p$ of a parallel algorithm using p processors is:

$$\mathcal{E}_p = \frac{\mathcal{S}_p}{\mathcal{S}_{max}}, \tag{2}$$

where

$$\mathcal{S}_{max} = \frac{p}{1 + f \times (p-1)} \tag{3}$$

is the maximum speedup a parallel algorithm can achieve and $0 < f < 1$ corresponds to the sequential portion of the parallel algorithm (i.e., the fraction that cannot be improved using parallelism). This comes from the Amdahl's law [12].

By increasing the problem size proportionally to the number of processors p, the scale-up refers to the ability of solving a problem p times larger in the same amount of time the corresponding sequential algorithm would solve a problem $1/p$ times lower and is defined as:

Definition 9. The *scale-up* U_p of a parallel algorithm using p processors is:

$$U_p = \frac{T_p}{T_1}, \tag{4}$$

where T_1 is the execution time of the sequential algorithm to solve a problem of size X and T_p is the execution time of the parallel algorithm with p processors to solve a problem of size pX.

3. SEQUENTIAL ALGORITHM

In this section we describe the sequential algorithm presented in [4]. It is a two-step algorithm. In the first step, it partitions the input key set into small buckets. This step is equivalent to an external multi-way merge sort carefully engineered to make it work in linear time. In the second step, it generates a MPHF for each bucket.

Figure 1 illustrates the two steps of the algorithm: the *partitioning step* and the *searching step*. The partitioning step takes a key set $S \subseteq \{0,1\}^L$ of size n and uses a hash function $h_0 : S \rightarrow \{0,1\}^b$ to partition S into $N_b = 2^b$ buckets for some integer $b \leq \log n + \log\log\log n - \log \ell + O(1)$,

where $\ell = \Omega(\log n \log\log n)$. The searching step generates a MPHF for each bucket i, $0 \leq i \leq N_b - 1$, and computes the *offset* array. The evaluation of the MPHF generated by the algorithm for a key x is:

$$MPHF(x) = MPHF_i(x) + offset[i] \quad (5)$$

where $i = h_0(x)$ is the bucket where key x is, $MPHF_i(x)$ is the position of x in bucket i, and $offset[i]$ gives the total number of entries before bucket i in the hash table.

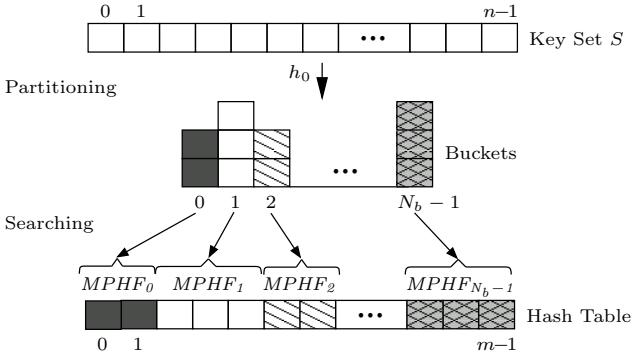


Figure 1: The two steps of the algorithm.

As mentioned before, the algorithm uses external memory to allow the construction of MPHFs for sets in the order of billion keys. The basic idea to obtain scalability is to partition the input key set into small buckets such that each bucket fits in the CPU cache – that is why it was called External Cache-Aware (ECA) algorithm.

Splitting the problem into small buckets has both theoretical and practical implications. From the theoretical point of view, Botelho, Pagh and Ziviani [3, 4] have shown how to simulate fully random hash functions on the small buckets, being able to prove that the ECA algorithm will work for every key set with high probability. From the practical point of view they have shown how to make buckets that are small enough to fit in the CPU cache, resulting in a significant speedup in processing time per element compared to other methods known in the literature.

The ECA algorithm is a randomized algorithm of Las Vegas type[2] because it uses in its second step the algorithm presented in [3] that works on random 3-partite hypergraphs[3], which is also a randomized algorithm of Las Vegas type. The ECA algorithm first scans the list of keys and computes the hash function values that will be needed later on in the algorithm. These values will (with high probability) distinguish all keys, so the original keys are discarded.

To form the buckets the hash values of the keys are sorted according to the value of h_0. In order to get scalability for large key sets, this is done using an implementation of an external memory mergesort [9] with some nuances to make it work in linear time. The total work on disk consists of reading the keys, plus writing and reading the hash function values once. Since the h_0 hash values are relatively small (less than 15 decimal digits) the radix sort is used to do the internal memory sorting of the runs.

Figure 2 presents a pseudo code for the ECA algorithm. The detailed description of the partitioning and searching steps are presented in Sections 3.1 and 3.2, respectively.

```
function ECA (S, H, {MPHF_0, ..., MPHF_{N_b-1}}, offset)
    Partitioning (S, H, Files)
    Searching (Files, {MPHF_0, ..., MPHF_{N_b-1}}, offset)
```

Figure 2: The ECA algorithm.

3.1 Partitioning Step

The partitioning step performs two important tasks. First, the variable-length keys are mapped to γ-bit signatures, which from now on will be called as *fingerprints*, by using a linear hash function $h' : S \to \{0,1\}^\gamma$ taken uniformly at random from the family $\mathcal{H}$ of linear hash functions (see [4] for details on $\mathcal{H}$). That is, the variable-length key set $S \subseteq \{0,1\}^L$ is mapped to a fixed-length key set F of fingerprints. To succeed with high probability γ was set to 96 bits or 12 bytes. Second, the set S of n keys is partitioned into N_b buckets, where b is a suitable parameter chosen to guarantee that each bucket has at most $\ell = \Omega(\log n \log\log n)$ keys with high probability (see [4] for details). It outputs a set of *Files* containing the buckets, which are merged in the searching step when the buckets are read from disk. Figure 3 presents the partitioning step.

```
function Partitioning (S, H, Files)
    ▶ Let β be the size in bytes of the fixed-length key
      set F
    ▶ Let μ be the size in bytes of an a priori reserved
      internal memory area
    ▶ Let N_f = ⌈β/μ⌉ be the number of key blocks that
      will be read from disk into an internal memory area

1. select h' uniformly at random from H
2. for j = 1 to N_f do
3.     DiskReader (S_j) {read a key block S_j from disk}
4.     Hashing (S_j, B_j) {store h'(x), for each x ∈ S_j,
                           into B_j, where |B_j| = μ}
5.     BucketSorter (B_j) {cluster B_j into N_b buckets using an
                           indirect radix sort algorithm that
                           takes h_0(x) for x ∈ S_j as sorting key
                           (i.e, the b most significant bits of
                           h'(x)) and if any bucket B_i has more
                           than ℓ keys restart partitioning step}
6.     BucketDumper (B_j, Files[j]) {dump B_j to disk into
                                     Files[j]}
```

Figure 3: Partitioning step.

Figure 4(a) shows a *logical* view of the N_b buckets generated in the partitioning step. In reality, the γ-bit fingerprints belonging to each bucket are distributed among many files, as depicted in Figure 4(b). In the example of Figure 4(b), the γ-bit fingerprints in bucket 0 appear in files 1 and N_f, the γ-bit fingerprints in bucket 1 appear in files 1, 2 and N_f, and so on.

This scattering of the γ-bit fingerprints in the buckets could generate a performance problem because of the potential number of seeks needed to read the γ-bit fingerprints in each bucket from the N_f files on disk during the second

[2]A random algorithm is Las Vegas if it always produces correct answers, but with a small probability of taking long to execute.

[3]A hypergraph is the generalization of a standard undirected graph where each edge connects $r \geq 2$ vertices.

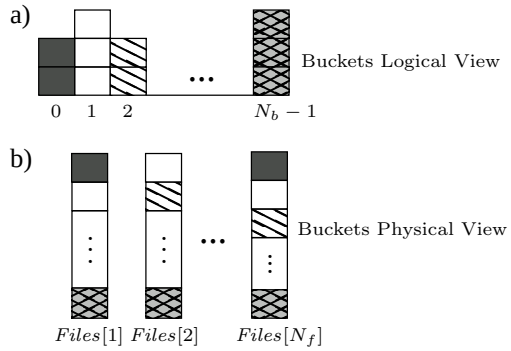


Figure 4: Situation of the buckets at the end of the partitioning step: (a) Logical view (b) Physical view.

step. But, as showed in [4], the number of seeks can be kept small by using buffering techniques.

3.2 Searching Step

Figure 5 presents the searching step. The searching step is responsible for generating a $MPHF_i$ for each bucket and for computing the *offset* array.

```
function Searching (Files, {MPHF_0, ..., MPHF_{N_b-1}}, offset)
    ▶ Let H be a minimum heap of size N_f
    ▶ Let the order relation in H be given by
      i = x[γ − b + 1, γ] for x ∈ F

1. for j = 1 to N_f do { Heap construction }
2.    Read the first γ-bit fingerprint x from Files[j]
      on disk
3.    Insert (i, j, x) in H
4. for i = 0 to N_b − 1 do
5.    BucketReader (Files, H, B_i) {Read bucket B_i from disk
                                    driven by heap H}
6.    if MPHFGen (B_i, MPHF_i) fails then
         Restart the partitioning step
7.    offset[i + 1] = offset[i] + |B_i|
8.    MPHFDumper (MPHF_i, offset[i]) {Write the description
                                      of MPHF_i and
                                      offset[i] to the disk}
```

Figure 5: Searching step.

Statement 1 of Figure 5 constructs a heap H of size N_f, which is well known to be linear on N_f. The order relation in H is given by the bucket address i (i.e., the b most significant bits of $x \in F$). Statement 4 has four steps. In statement 5, a bucket is read from disk, as described below. In statement 6, a function $MPHF_i$ is generated for each bucket B_i using an algorithm based on 3-partite random hypergraphs presented in [3]. In statement 7, the next entry of the *offset* array is computed. Finally, statement 8 writes the description of $MPHF_i$ and $offset[i]$ to disk. Note that to compute $offset[i+1]$ we just need $|B_i|$ (i.e., the number of keys in bucket B_i) and $offset[i]$. So, we just need to keep two entries of the *offset* array in memory all the time.

The algorithm to read bucket B_i from disk is presented in Figure 6. Bucket B_i is distributed among many files and the heap H is used to drive a multiway merge operation. Statement 2 extracts and removes triple (i, j, x) from H, where i is a minimum value in H. Statement 3 inserts x in bucket B_i. Statement 4 performs a seek operation in $Files[j]$ on disk for the first read operation and reads sequentially all γ-bit fingerprints $x \in F$ that have the same index i and inserts them all in bucket B_i. Finally, statement 5 inserts in H the triple (i', j, x'), where $x' \in F$ is the first γ-bit fingerprint read from $Files[j]$ (in statement 4) that does not have the same bucket address as the previous keys.

```
function BucketReader (Files, H, B_i)
1. while bucket B_i is not full do
2.    Remove (i, j, x) from H
3.    Insert x into bucket B_i
4.    Read sequentially all γ-bit fingerprints from Files[j]
      that have the same i and insert them into B_i
5.    Insert the triple (i', j, x') in H, where x' is the first
      γ-bit fingerprint read from Files[j] that does not
      have the same bucket index i
```

Figure 6: Reading a bucket.

4. DISTRIBUTED AND PARALLEL ALGORITHM

In this section we describe our *Parallel External Cache-Aware* (PECA) algorithm. As mentioned before, the main motivation for implementing a distributed and parallel version of the ECA algorithm is scalability in terms of the size of the key set that has to be processed. In this case, we must assume that the keys to be processed will be distributed among several machines. Further, both the buckets and the construction of the hash functions for each bucket are also distributed among the participating machines. In this scenario, the partitioning and the searching steps present different requirements when compared to the sequential version, as we discuss next.

In Section 4.1 we discuss how to speedup the construction of a MPHF by distributing the buckets (during the partitioning phase) and the construction of the MPHF$_i$ for each bucket (during the searching phase) among the participating machines. In Section 4.2 we present a version of the PECA algorithm where both the description and the evaluation of the MPHF obtained is *centralized* in one machine, from now on referred to as *PECA-CE*. In Section 4.3 we present another version of the PECA algorithm where both the description and the evaluation of the MPHF obtained is *distributed* among the participating machines, from now on referred to as *PECA-DE*.

4.1 Distributed Construction of MPHFs

In this section we present the steps that are common to both PECA-CE and PECA-DE algorithms. We employed two types of processes: manager and worker. This scheme is shown in Figure 7.

The manager is responsible for assigning tasks to the workers, determining global values during the execution, and dumping the resulting MPHFs received from the workers to disk. This last task is different for the PECA-CE and PECA-DE algorithms, as we will show later on.

The worker stores a partition of the key set, its buckets and the related MPHF of each bucket. Each worker sends and receives data from other workers whenever necessary. The workers are implemented as thread-based processes, where each thread is responsible for a task, allowing larger overlap between computation and communication (disk and network) in both steps of the algorithm.

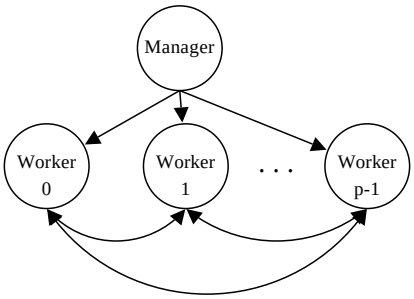


Figure 7: The manager/worker scheme.

Our major challenge in producing such a distributed version is that we do not know in advance which keys will be clustered together in the same bucket. Our strategy in this case is to migrate data whenever necessary. On the other hand, once we have the buckets, we are able to generate the MPHFs.

The manager starts the processing by sending the overall assignment of buckets to workers before each worker starts processing its portion of the keys, so that each worker becomes aware of the worker to which keys (actually, fingerprints) must be sent. For that verification, the manager sends the following information: (i) the function $h' \in \mathcal{H}$ used to compute the fingerprints; (ii) the worker identifier i, where $0 \leq i < p$ and p is the number of workers; and (iii) the number of buckets per worker, which is given by $B_{pw} = \lceil N_b/p \rceil$ (recall that N_b is the number of buckets). Therefore, each worker i is responsible for the buckets in the range $[iB_{pw}, (i+1)B_{pw} - 1]$.

Each worker then starts reading a key $k \in S$, applies the received hash function h' and verifies whether it belongs to another worker. For that each worker i computes $w = h_0(k)/B_{pw}$ and checks if $w \neq i$ (recall that $h_0(k)$ corresponds to the b most significant bits of $h'(k)$.) If it is the case, it sends the corresponding fingerprint to the worker w, otherwise, it stores the fingerprint locally for further processing.

Figure 8 illustrates the partitioning step in each worker. The partitioning step of the sequential algorithm presented in Figure 3 is divided into four major tasks: data reading (line 3), hashing (line 4), bucket sorting (line 5), and bucket dumping (line 6).

As depicted in Figure 8, the worker is divided into the following six threads:

1. *Disk Reader*: it reads the keys from the worker's portion of the set S and puts them in Queue 1. When there are no more keys to be read, then an end of file marker is put in Queue 1.

2. *Hashing*: it gets the keys from Queue 1 and generates the fingerprints for the keys, as mentioned in Section 3. This thread then checks whether the key being currently analyzed is assigned to another worker. If it is, its fingerprint is passed to the Sender thread through Queue 5, otherwise its fingerprint is placed in Queue 2. When there are no more keys to be processed in Queue 1, then an end of file marker is put in both Queue 2 and 5.

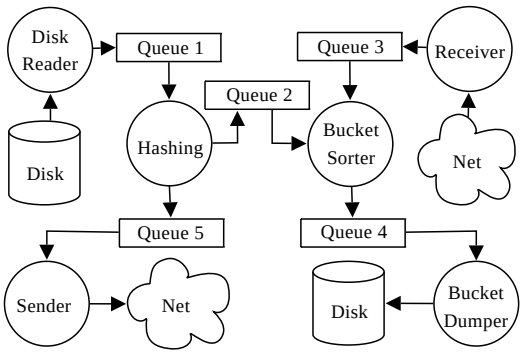


Figure 8: The partitioning step in the worker.

3. *Sender*: it sends a fingerprint taken from Queue 5 to the worker that is responsible for it. When there are no more fingerprints in Queue 5, then an end of file marker is sent to all other workers.

4. *Receiver*: it receives fingerprints sent from other workers through the net, and puts them in Queue 3. It finishes its work when an end of file marker is received from all other workers.

5. *Bucket Sorter*: it takes fingerprints from Queues 2 and 3 until a buffer of size $\mu/2$ bytes is completely full (recall that μ is the amount of internal memory available), organizing them into buckets, and puts them in Queue 4. The process is repeated until an end of file marker is obtained from both Queues 2 and 3. In this case, it also places an end of file marker in Queue 4.

6. *Bucket Dumper*: it takes the buckets from Queue 4 and writes them to disk, for further processing by the searching step. It finishes when an end of file marker is taken from Queue 4.

After each worker finishes the partitioning step, it sends the size of each bucket to the manager, which then calculates the offset array. This does not depend on the searching step, so the manager may compute the offset array whereas the workers are performing the searching step.

Figure 9 illustrates the searching step in each worker. It consists of generating the functions $MPHF_i$ for each bucket i. The searching step of the sequential algorithm of Figure 5 is divided into three tasks: bucket reading (line 5), MPHF construction (lines 6 and 7), and MPHF dumping (line 8). Notice that, in this step, there is no need for communication between workers, since the generation of the $MPHF_i$ for each bucket does not depend on keys that are in other buckets.

Again, the worker is divided into threads of execution, each thread being responsible for a task. Following Figure 9, the worker is divided into the following two threads:

1. *Bucket Reader*: it reads the buckets from disk, and puts them in Queue 1. When there are no more buckets to be read, then an end of file marker is put in Queue 1.

2. *MPHF Gen*: it gets buckets from Queue 1 and generates the functions for them until no more bucket remains. It can be instantiated t times, where t can be thought of as the number of processors of the machine.

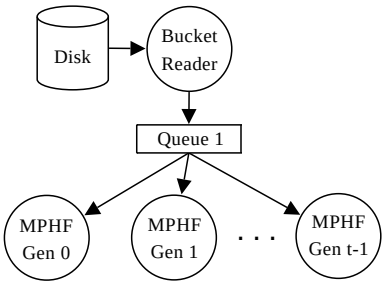


Figure 9: The searching step in the worker.

4.2 Centralized Evaluation of the MPHF

In this section we present the PECA-CE algorithm, where both the description and the evaluation of the MPHF is centralized in a single machine (the one running the manager process).

After each worker finishes the partitioning step, it sends the size of each bucket to the manager, which then calculates the offset array. This does not depend on the searching step, so the manager may compute the offset array whereas the workers are performing the searching step. After each worker finishes the construction of the MPHFs of their buckets, it sends them to the manager, that will then write sequentially the final MPHF to disk, and the algorithm resumes.

The task of writing the final MPHF to disk corresponds to the sequential part of the algorithm and represents approximately 0.5% of the execution time. Thus, there is a fraction of 99.5% of the execution time from which we can exploit parallelism. That is why the PECA-CE algorithm can be considered an embarrassingly parallel algorithm.

The evaluation of the resulting MPHF is done in the same way as it is done in the sequential algorithm presented in Section 3 (see Eq. (5)).

4.3 Distributed Evaluation of the MPHF

In this section we present the PECA-DE algorithm, where both the description and the evaluation of the MPHF are distributed and stored locally in each worker. The PECA-DE algorithm calculates a *localoffset* array in each worker, in the same way as it is done in the searching step of the sequential algorithm shown in Figure 5 (see line 7). At the end of the partitioning step, each worker sends the number of keys assigned to it to the manager, which calculates a *globaloffset*, whereas the workers are performing the searching step.

To evaluate a key k using the resulting MPHF, the manager first discovers the worker w that generated the MPHF for the bucket in which k is (recall that this is done by calculating $w = h_0(k)/B_{pw}$). Then, the key k (actually, its fingerprint) is sent to the worker w, which calculates locally a partial result

$$MPHF_{partial}(k) = MPHF_i(k) + localoffset[i],$$

where $i = h_0(k) \bmod B_{pw}$ is the local bucket address where k belongs and $localoffset[i]$ gives the total number of keys before bucket i. Once this partial result is calculated, it is sent back to the manager, which calculates the final result

$$MPHF(k) = MPHF_{partial}(k) + globaloffset[w],$$

where $globaloffset[w]$ has p entries and gives the total number of keys handled by the workers before worker w.

The downside of this is that the evaluation of a single key is harmed, due to the communication overhead between the manager and the workers. However, if the system is being fed by a key stream, the average performance will improve because p keys can be evaluated in parallel by p workers. This will indeed happen because the keys are uniformly placed in the buckets by using a hash function, which will balance the key stream among the p workers. The experimental results in Section 5 confirm this fact.

Other advantage of the PECA-DE algorithm is that the workers do not need to send the MPHFs generated locally for the buckets they are responsible for to the manager. Instead, they are written in parallel by the workers. Therefore, in this case, the fraction of parallelism we can potentially exploit corresponds to 100% of the execution time.

Therefore, as shown in Section 5, the PECA-DE algorithm provides a slightly better construction time than the PECA-CE algorithm. But the main advantage of the PECA-DE algorithm is that it distributes the resulting MPHF among several machines. When the number n of keys in the key set S grows, the size of the resulting MPHF also grows linearly with n. For very large n, it may not be possible to represent the resulting MPHF in just one machine, whereas the PECA-DE algorithm addresses this by distributing uniformly the resulting MPHF.

4.4 Implementation Decisions

In this section we present and discuss some implementation decisions that aim to reduce the overhead of the distributed algorithms we just described.

A very first decision is to exploit multiprogramming in the worker, motivated not only by the characteristics of the execution platform, but also by the complementary profiles of the steps, which are either CPU or I/O-intensive. As a result, we are able to maximize the overlap between computation and communication, represented by disk and network traffic.

Further, in order to reduce the overhead due to context changes we grouped steps (described in Section 4) into fewer threads, as detailed next. This strategy speeds up the execution time, even on a single core machine, which is our case.

In the partitioning step, the *Hashing* and *Bucket Sorter* threads were grouped together into a single thread, as shown in Figure 10. Notice that these two steps are the most CPU-intensive and the merge would prevent them to contend for the CPU. As a result, one thread is almost always keeping the CPU busy, while the remaining threads are usually waiting for system calls to resume (*Disk Reader* reading data from disk, *Net Reader* receiving messages from the net, and *Bucket Dumper* writing buckets back to disk whenever necessary).

In the searching step, the structure replicates the step-based division presented, but instantiating just one MPHF Gen thread (i.e., $t = 1$), as shown in Figure 11.

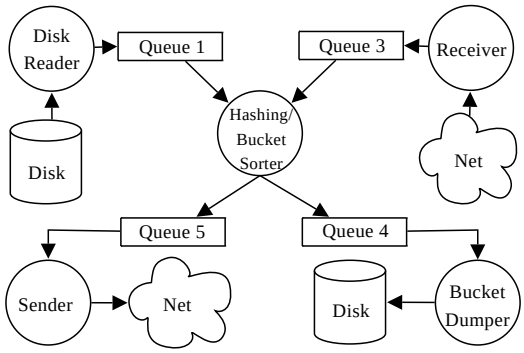


Figure 10: The actual partitioning step used in the experiments.

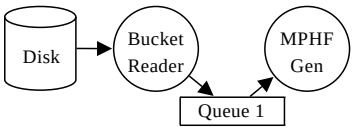


Figure 11: The actual searching step used in the experiments.

We also coalesced messages for both reducing the number of system calls associated with exchange messages and better exploiting the available bandwidth. That is, we group the fingerprints that were going to be sent from one to another worker in buffers of a fixed size.

5. EXPERIMENTAL RESULTS

The purpose of this section is to evaluate the performance of both the PECA-CE and PECA-DE algorithms in terms of speedup and scale-up (see Definitions 7 and 9), considering the impact of the key size in both metrics. We also verify whether the load is balanced among the workers. To compute the metrics we use the time to construct a MPHF in the distributed algorithms.

The experiments were run in a cluster with 14 equal single core machines, each one with 2.13 gigahertz, 64-bit architecture, running the Linux operating system version 2.6, and 2 gigabytes of main memory.

For the experiments we used three collections: (i) a set of URLs collected from the web, (ii) a set of randomly generated 16-byte integers, and (iii) a set of randomly generated 8-byte integers. The collections are presented in Table 1. The main reason to choose these three different collections is to evaluate the impact of the key size on the results.

Collection	Average key size	n (billions)
URLs	64	1.024
Random Integers	16	1.024
	8	1.024

Table 1: Collections used for the experiments.

In Section 5.1 we discuss the impact of key size on speedup and scale-up. In Section 5.2 we study the communication overhead. In Section 5.3 we discuss the load balance among workers. In Section 5.4 we discuss the distributed evaluation of a MPHF when the MPHF is being fed by a key stream.

5.1 Key Size Impact

In this section we evaluate the impact of the key size and how it changes as we increase the number of processors. We use both speedup and scale-up as metrics for performing such evaluation.

In order to compute the speedup we need the execution time of the sequential ECA algorithm. Table 2 shows how much time the ECA algorithm requires to build a MPHF for 1.024 billion keys taken from each collection shown in Table 1.

n (billion)	Collection	time (min.)
1.024	64-byte URLs	50.02
	16-byte integers	39.35
	8-byte integers	34.58

Table 2: Time in minutes of the sequential algorithm (ECA) to construct a MPHF for 1.024 billion keys.

We start by evaluating the speedup of the distributed algorithm and perform three sets of experiments, using the three collections presented in Table 1 and varying the number of machines from 1 to 14.

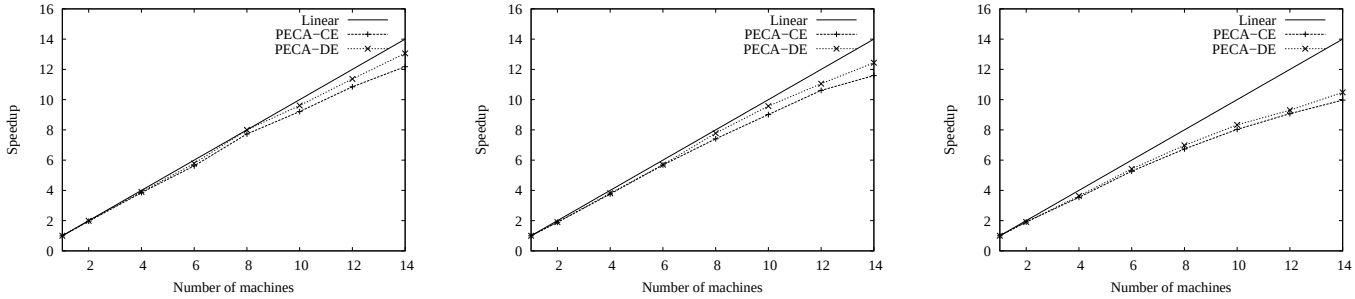
Table 3 presents the maximum speedup ($\mathcal{S}_{max}$), the speedup $\mathcal{S}_p$ and the efficiency $\mathcal{E}_p$ for both the PECA-CE and PECA-DE algorithms for each collection. In almost all cases, the speedup was very good, achieving an efficiency of up to 93% using 14 machines, confirming the expectations of that not only there is a parallelism opportunity to be exploited, but also it is significative enough that allows good efficiencies even for relatively large configurations. The comparison between PECA-CE and PECA-DE also shows that the strategy employed in PECA-DE was effective.

It is remarkable that the key size impacts the observed speedups, since the efficiency for the 64-byte URLs is greater than 90% for all configurations evaluated, but for 16-byte and 8-byte random integers it is greater than or equal to 90% only for $p \geq 12$ and $p \geq 6$, respectively. This happens because when we decrease the key size, the amount of computation decreases proportionally in the partitioning step, but the amount of communication remains constant since the γ-bit fingerprints will continue with the same size $\gamma = 96$ bits (or 12 bytes.) The size γ of a fingerprint depends on the number of keys n, but does not depend on the key size [4]. Therefore, the smaller is the key size, the smaller is the value of p to fully exploit the available parallelism, resulting in eventual performance degradation. A graphical view of the speedups can also be seen in Figure 12.

We performed similar sets of experiments for evaluating the scale-up and the results are presented in Table 5 and Figure 13, where we may confirm the good scalability of the algorithm, which allows just 17% of degradation when using 14 machines to solve a problem 14 times larger. These results show that not only the algorithm proposed is efficient, but also is very effective dealing with larger datasets. For instance, in Table 4 it is shown that the performance degradation is up to 20% even for 14.336 billion keys evenly distributed among 14 machines. Again, the key size has a definite impact on the performance.

p	$\mathcal{S}_{max}$		64-byte URLs				16-byte random integers				8-byte random integers			
			PECA-CE		PECA-DE		PECA-CE		PECA-DE		PECA-CE		PECA-DE	
	PECA-CE	PECA-DE	$\mathcal{S}_p$	$\mathcal{E}_p$	$\mathcal{S}_p$	$\mathcal{E}_p$	$\mathcal{S}_p$	$\mathcal{E}_p$	$\mathcal{S}_p$	$\mathcal{E}_p$	$\mathcal{S}_p$	$\mathcal{E}_p$	$\mathcal{S}_p$	$\mathcal{E}_p$
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	1.99	2.00	1.96	0.98	1.99	1.00	1.89	0.95	1.90	0.95	1.91	0.96	1.91	0.96
4	3.94	4.00	3.85	0.98	3.90	0.98	3.76	0.95	3.81	0.95	3.54	0.90	3.63	0.91
6	5.85	6.00	5.62	0.96	5.78	0.96	5.68	0.97	5.70	0.95	5.27	0.90	5.42	0.90
8	7.73	8.00	7.73	1.00	8.00	1.00	7.41	0.96	7.78	0.97	6.74	0.87	6.98	0.87
10	9.57	10.00	9.21	0.96	9.61	0.96	9.01	0.94	9.57	0.96	8.03	0.84	8.33	0.83
12	11.37	12.00	10.85	0.95	11.37	0.95	10.61	0.93	11.05	0.92	9.07	0.80	9.30	0.78
14	13.15	14.00	12.18	0.93	13.06	0.93	11.59	0.88	12.44	0.89	9.97	0.76	10.48	0.75

Table 3: Speedup obtained with a confidence level of 95% for both the PECA-CE and PECA-DE algorithms considering 1.024 billion keys (73,142,857 keys in each machine).



(a) 64-byte URLs (b) 16-byte random integers (c) 8-byte random integers

Figure 12: Speedup obtained with a confidence level of 95% for both the PECA-CE and PECA-DE algorithms considering 1.024 billion keys (73,142,857 keys in each machine).

n (billions)	Random integer collections	Construction time (min)		
		ECA	PECA-DE	U_p
14.336	16-byte	41.17	49.5	1.20
	8-byte	34.58	58.00	1.68

Table 4: Scale-up obtained with a confidence level of 95% for the PECA-DE algorithm considering 14.336 billion keys (1.024 billion keys in each machine).

5.2 Communication Overhead

We now analyze the communication overhead. There is a significant overhead associated with message traffic among workers in the net. Since the hash function h_0 is a linear hash function [4] that behaves closely to a fully random hash function, the chance of a given key in the key set S belonging to a given bucket is close to $\frac{1}{N_b}$. Since each worker has $\frac{N_b}{p}$ buckets, the chance that a key it reads belongs to another worker is close to $\frac{p-1}{p}$. Since each worker has to read $\frac{n}{p}$ keys from disk, it will send through the net approximately

$$\frac{n(p-1)}{p^2}.$$

Thus, the total traffic τ of fingerprints through the net is approximately

$$\tau \approx \frac{n(p-1)}{p}. \quad (6)$$

Table 6 shows the minimum and maximum amount of keys sent to the net by a worker. It also shows the expected amount computed by using Eq. (6). As it shows, the empirical measurements are really close to the expected value.

p	Keys sent by a worker to the net		
	Max (%)	Min (%)	τ (%)
2	50.005	49.996	50.000
4	75.008	74.994	75.000
6	83.339	83.327	83.333
8	87.506	87.492	87.500
10	90.009	89.991	90.000
12	91.673	91.657	91.667
14	92.864	92.849	92.857

Table 6: Worst, best and expected percentage of keys sent by a worker to the net.

That results in a relevant overhead due to communication among the workers, and as the number of workers increases, the speedup can be penalized if the network bandwidth is not enough for the traffic. In our 1 gigabit ethernet network this was not a problem for at most 14 workers.

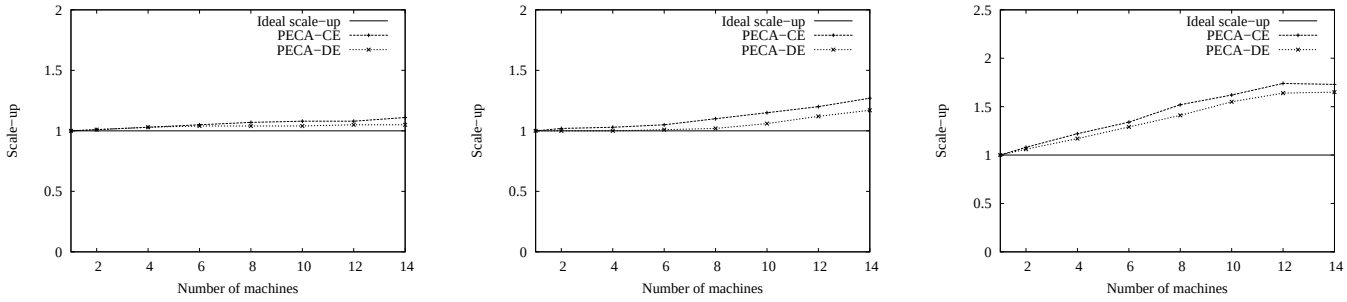
5.3 Load Balancing

In this section we quantify the load imbalance and correlate it with the results. An important issue is how much the load is balanced among the workers. The load depends on the following parameters: (i) the number of keys each worker reads from disk in the partitioning step; (ii) the number of buckets each worker is responsible for; (iii) the number of keys in each bucket.

The first two parameters are fixed by construction and are evenly distributed among the workers. The only parameter that could present some variation in each execution is the

p	64-byte URLs				16-byte random integers				8-byte random integers			
	PECA-CE		PECA-DE		PECA-CE		PECA-DE		PECA-CE		PECA-DE	
	time (min)	U_p	time (min)	U_p	time (min)	U_p	time (min)	U_p	time (min)	U_p	time (min)	U_p
1	3.71	1.00	3.68	1.00	2.68	1.00	2.70	1.00	2.00	1.00	2.00	1.00
2	3.76	1.01	3.71	1.01	2.74	1.02	2.69	1.00	2.16	1.08	2.11	1.06
4	3.84	1.03	3.77	1.03	2.77	1.03	2.71	1.00	2.44	1.22	2.35	1.17
6	3.91	1.05	3.81	1.04	2.82	1.05	2.73	1.01	2.68	1.34	2.58	1.29
8	3.96	1.07	3.82	1.04	2.94	1.10	2.76	1.02	3.04	1.52	2.82	1.41
10	4.02	1.08	3.83	1.04	3.10	1.15	2.86	1.06	3.25	1.62	3.10	1.55
12	4.02	1.08	3.84	1.05	3.23	1.20	3.02	1.12	3.48	1.74	3.29	1.64
14	4.11	1.11	3.85	1.05	3.40	1.27	3.16	1.17	3.47	1.73	3.30	1.65

Table 5: Scale-up obtained with a confidence level of 95% for both the PECA-CE and PECA-DE algorithms considering 1.024 billion keys (73,142,857 keys in each machine).



(a) 64-byte URLs (b) 16-byte random integers (c) 8-byte random integers

Figure 13: Scale-up obtained with a confidence level of 95% for both the PECA-CE and PECA-DE algorithms considering 1.024 billion keys (73,142,857 keys in each machine).

last one. However, as we use the hash function h_0 to split the key set into buckets, it was shown in [4] that each key goes to a given bucket with probability close to $1/N_b$ and therefore the distribution of the bucket sizes follows a binomial distribution with average n_p/B_{pw}, where $n_p = \sum_{i=0}^{B_{pw}-1} |B_i|$ is the number of keys each worker has stored in the buckets it is responsible for and B_{pw} is the number of buckets per worker.

It is also shown in [4] that the largest bucket is within a factor $O(\log \log n_p)$ of the average bucket size. Therefore, n_p has a very small variation from worker to worker, which makes the load balanced among the p machines. Table 7 presents experimental results confirming this, as the difference between the execution time of the fastest worker (t_{fw}) and the slowest worker (t_{sw}) was less than or equal to 0.1 minutes.

p	PECA-CE			PECA-DE		
	t_{fw}	t_{sw}	$t_{sw}-t_{fw}$	t_{fw}	t_{sw}	$t_{sw}-t_{wb}$
2	25.27	25.37	0.10	25.01	25.06	0.05
4	12.94	13.04	0.10	12.78	12.86	0.09
6	8.58	8.69	0.11	8.53	8.65	0.11
8	6.19	6.26	0.07	6.18	6.25	0.07
10	5.14	5.22	0.08	5.11	5.20	0.09
12	4.31	4.39	0.08	4.32	4.40	0.07
14	3.81	3.88	0.07	3.76	3.84	0.08

Table 7: Fastest worker time (t_{fw}), slowest worker time (t_{sw}), and difference between t_{sw} and t_{fw} to show the load balancing among the workers for 1.024 billion 64-byte URLs distributed in p machines. The times are in minutes.

5.4 Distributed Evaluation

In this section we show that the distributed evaluation of a MPHF is worth when compared to the ones generated by both the sequential and PECA-CE algorithms. These results assume that the distributed function is being fed by a key stream, instead of one key at a time.

Table 8 shows the times that both the ECA algorithm and PECA-DE algorithm needs to evaluate one billion keys taken at random. As expected, the distributed evaluation was faster because p keys of the key stream can be evaluated in parallel by p participating machines. Here we also used the message coalescing technique.

Collection	Evaluation time (min)	
	ECA	PECA-DE
64-byte URLs	33.11	21.68
16-byte random integers	24.54	11.47
8-byte random integers	18.2	10.1

Table 8: Evaluation time in minutes for both the sequential algorithm ECA and the parallel algorithm PECA-DE algorithm, considering 1 billion keys.

6. CONCLUSIONS

In this paper we have presented a parallel implementation of the External Cache-Aware (ECA) perfect hashing algorithm presented in [4]. We have designed two versions. The PECA-CE algorithm distributes the construction of the resulting MPHFs among p machines and centralize the evaluation and description of the resulting functions in a single

machine, as in the sequential case. Then the goal in this version is to speedup the construction of the MPHFs by exploiting the high degree of parallelism of the ECA algorithm. The PECA-DE algorithm distributes both the construction and the evaluation of the resulting MPHFs. In this version the goal is to allow the descriptions of the resulting functions be uniformly distributed among the participating machines.

We have evaluated both the PECA-CE and PECA-DE algorithms using speedup and scale-up as metrics. Both versions presented an almost linear speedup, achieving an efficiency larger than 90% by using 14-computer cluster and keys of average size larger than or equal to 16 bytes. For smaller keys, e.g. 8-byte integers, we have shown that the existent parallelism between computation and communication is captured with 90% of efficiency by using a smaller number of machines (e.g, $p = 6$). This was as expected, because the smaller is the key the smaller is the amount of computation, but the amount of communication remains constant for a given number n of keys, penalizing the speedup.

We have also shown that both the PECA-CE and PECA-DE algorithms scale really well for larger keys. Smaller keys also impose restrictions on the scalability due to the smaller degree of overlap between computation and communication aforementioned. To illustrate the scalability, the time to generate a MPHF for 14.336 billion 16-byte random integers using a 14-computer cluster with 1.024 billion 16-byte random integers in each machine is just a factor of 1.2 more than the time spent by the sequential algorithm when applied to 1.024 billion keys.

7. ACKNOWLEDGMENTS

We thank the partial support given by GERINDO Project Grant MCT/CNPq/CT-INFO 552087/2002-5, INFOWEB Project Grant MCT/CNPq/CT-INFO 550874/2007-0, FAPEMIG Project Grant CEX 1347/05, and CNPq Grants 30.5237/02-0 (Nivio Ziviani), 484744/2007-0 (Wagner Meira Jr.), 312517/2006-8 (Wagner Meira Jr.) and 380792/2008-7 (Fabiano C. Botelho).

8. REFERENCES

[1] P. Boldi and S. Vigna. The webgraph framework i: Compression techniques. In *Proceedings of the 13th International World Wide Web Conference (WWW'04)*, pages 595–602, 2004.

[2] F. Botelho, Y. Kohayakawa, and N. Ziviani. A practical minimal perfect hashing method. In *Proceedings of the 4th International Workshop on Efficient and Experimental Algorithms (WEA'05)*, pages 488–500. Springer LNCS vol. 3503, 2005.

[3] F. Botelho, R. Pagh, and N. Ziviani. Simple and space-efficient minimal perfect hash functions. In *Proceedings of the 10th Workshop on Algorithms and Data Structures (WADs'07)*, pages 139–150. Springer LNCS vol. 4619, 2007.

[4] F. Botelho and N. Ziviani. External perfect hashing for very large key sets. In *Proceedings of the 16th ACM Conference on Information and Knowledge Management (CIKM'07)*, pages 653–662. ACM Press, 2007.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th International World Wide Web Conference (WWW'98)*, pages 107–117, April 1998.

[6] C.-C. Chang and C.-Y. Lin. A perfect hashing schemes for mining association rules. *The Computer Journal*, 48(2):168–179, 2005.

[7] C.-C. Chang, C.-Y. Lin, and H. Chou. Perfect hashing schemes for mining traversal patterns. *Journal of Fundamenta Informaticae*, 70(3):185–202, 2006.

[8] A. M. Daoud. Perfect hash functions for large web repositories. In G. Kotsis, D. Taniar, S. Bressan, I. K. Ibrahim, and S. Mokhtar, editors, *Proceedings of the 7th International Conference on Information Integration and Web Based Applications Services (iiWAS'05)*, volume 196, pages 1053–1063. Austrian Computer Society, 2005.

[9] P. Larson and G. Graefe. Memory management during run generation in external sorting. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 472–483. ACM Press, 1998.

[10] S. Lefebvre and H. Hoppe. Perfect spatial hashing. *ACM Transactions on Graphics*, 25(3):579–588, 2006.

[11] B. Prabhakar and F. Bonomi. Perfect hashing for network applications. In *Proceedings of the IEEE International Symposium on Information Theory*. IEEE Press, 2006.

[12] M. J. Quinn. *Parallel computing: theory and practice*. McGraw-Hill, Inc., New York, NY, USA, 1994.