A Highly Scalable Perfect Hashing Algorithm

Nivio Ziviani
Fabiano C. Botelho
Department of Computer Science
Federal University of Minas Gerais, Brazil

3rd Intl. Conference on Scalable Information Systems
Naples, Italy, June 4-6, 2008
Where is Belo Horizonte?
Pampulha’s Church
Oscar Niemeyer
Objective of the Presentation

Present a perfect hashing algorithm:

- Sequential construction of the function
- Distributed construction of the function
- Description and evaluation of the function:
  - *Centralized* in one machine
  - *Distributed* among the participating machines

Algorithm is highly scalable, time efficient and near space-optimal
Perfect Hash Function

Static key set $S$ of size $n$

Hash Table

$S \subseteq U$, where $|U| = u$
Minimal Perfect Hash Function

$S \subseteq U$, where $|U| = u$
The Algorithm

A perfect hashing algorithm that uses the idea of partitioning the input key set into small buckets:

- Key set fits in the internal memory
  - Internal Random Access memory algorithm
- Key set larger than the internal memory
  - External Cache-Aware memory algorithm
Where to use a PHF or a MPHF?

- Access items based on the value of a key is ubiquitous in Computer Science

- Work with huge static item sets:
  - In data warehousing applications:
    - On-Line Analytical Processing (OLAP) applications
  - In Web search engines:
    - Large vocabularies
    - Map long URLs in smaller integer numbers that are used as IDs
Indexing: Representing the Vocabulary

Collection of documents

Doc 1
Doc 2
Doc 3
Doc 4
Doc 5
...
Doc n

Vocabulary

Term 1
Term 2
Term 3
Term 4
Term 5
Term 6
Term 7
Term 8
...
Term t

Inverted List

Doc 1 | Doc 5 | ...
Doc 1 | Doc 2 | ...
Doc 3 | Doc 4 | ...
Doc 7 | Doc 9 | ...
Doc 6 | Doc 10 | ...
Doc 1 | Doc 5 | ...
Doc 9 | Doc 11 | ...

LATIN - LAboratory for Treating INformation (www.dcc.ufmg.br/latin)
Mapping URLs to Web Graph Vertices

<table>
<thead>
<tr>
<th>URLS</th>
<th>Web Graph Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>URL 1</td>
<td>0</td>
</tr>
<tr>
<td>URL 2</td>
<td>1</td>
</tr>
<tr>
<td>URL 3</td>
<td>2</td>
</tr>
<tr>
<td>URL 4</td>
<td>3</td>
</tr>
<tr>
<td>URL 5</td>
<td>4</td>
</tr>
<tr>
<td>URL 6</td>
<td>5</td>
</tr>
<tr>
<td>URL 7</td>
<td>6</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>URL n</td>
<td>n-1</td>
</tr>
</tbody>
</table>
Mapping URLs to Web Graph Vertices

URLS

URL 1
URL 2
URL 3
URL 4
URL 5
URL 6
URL 7
...
URL n

MPHF

Web Graph Vertices

0
1
2
3
4
5
6
n-1

LATIN - LAboratory for Treating INformation (www.dcc.ufmg.br/latin)
Information Theoretical Lower Bounds for Storage Space

- PHFs ($m \approx n$): Storage Space $\geq \frac{n^2}{m} \log e$

- MPHFs ($m = n$): Storage Space $\geq n \log e$

$m < 3n$

$\log e \approx 1.4427$
Uniform Hashing Versus Universal Hashing

Key universe $U$ of size $u$ \hspace{2cm} \text{Hash function} \hspace{2cm} \text{Range } M \text{ of size } m
Uniform Hashing Versus Universal Hashing

**Uniform hashing**

- # of functions from U to M?
  \[ m^u \]

- # of bits to encode each function
  \[ u \log m \]

- Independent functions with values uniformly distributed
Uniform Hashing Versus Universal Hashing

Key universe
$U$ of size $u$

Hash function

Range $M$ of size $m$

Uniform hashing

- # of functions from $U$ to $M$?
  \[ m^u \]

- # of bits to encode each function
  \[ u \log m \]

- Independent functions with values uniformly distributed

Universal hashing

- A family of hash functions $\mathcal{H}$ is universal if:
  - for any pair of distinct keys $(x_1, x_2)$ from $U$ and
  - a hash function $h$ chosen uniformly from $\mathcal{H}$ then:
    \[ \Pr(h(x_1) = h(x_2)) \leq \frac{1}{m} \]
Intuition Behind Universal Hashing

- We often lose relatively little compared to using a completely random map (uniform hashing)
- If S of size n is hashed to $n^2$ buckets, with probability more than $\frac{1}{2}$, no collisions occur
  - Even with complete randomness, we do not expect little $o(n^2)$ buckets to suffice (the birthday paradox)
  - So nothing is lost by using a universal family instead!
Related Work

- Theoretical Results
  (use uniform hashing)

- Practical Results
  (use universal hashing - assume uniform hashing for free)

- Heuristics
### Theoretical Results

Use Complete Randomness (Uniform Hash Functions)

<table>
<thead>
<tr>
<th>Work</th>
<th>Gen. Time</th>
<th>Eval. Time</th>
<th>Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hagerup and Tholey (2001)</td>
<td>$O(n+\log \log u)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
Theoretical Results
Use Complete Randomness (Uniform Hash Functions)

<table>
<thead>
<tr>
<th>Work</th>
<th>Gen. Time</th>
<th>Eval. Time</th>
<th>Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hagerup &amp; Tholey (2001)</td>
<td>O(n+log log u)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Botelho &amp; Ziviani (CIKM 2007)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
**Practical Results**

Assume Uniform Hashing for Free (Use Universal Hashing)

<table>
<thead>
<tr>
<th>Work</th>
<th>Gen. Time</th>
<th>Eval. Time</th>
<th>Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Czech, Havas &amp; Majewski (1992)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n log n)</td>
</tr>
<tr>
<td>Majewski, Wormald, Havas &amp; Czech (1996)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n log n)</td>
</tr>
</tbody>
</table>
### Practical Results

**Assume Uniform Hashing for Free (Use Universal Hashing)**

<table>
<thead>
<tr>
<th>Work</th>
<th>Gen. Time</th>
<th>Eval. Time</th>
<th>Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Czech, Havas &amp; Majewski (1992)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Majewski, Wormald, Havas &amp; Czech (1996)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Botelho, Pagh, Ziviani (WADs 2007)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
# Practical Results

Assume Uniform Hashing for Free (Use Universal Hashing)

<table>
<thead>
<tr>
<th>Work</th>
<th>Gen. Time</th>
<th>Eval. Time</th>
<th>Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Czech, Havas &amp; Majewski (1992)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Majewski, Wormald, Havas &amp; Czech (1996)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Botelho, Pagh, Ziviani (WADs 2007)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Botelho &amp; Ziviani (CIKM 2007)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
## Empirical Results

<table>
<thead>
<tr>
<th>Work</th>
<th>Application</th>
<th>Gen. Time</th>
<th>Eval. Time</th>
<th>Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fox, Chen &amp; Heath (1992)</td>
<td>Index data in CD-ROM</td>
<td>Exp.</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Lefebvre &amp; Hoppe (2006)</td>
<td>Sparse spatial data</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
The Sequential
External Cache-Aware Algorithm...
External Cache-Aware Memory Algorithm

- First MPHF algorithm for very large key sets (in the order of billions of keys)
- This is possible because
  - Deals with external memory efficiently
  - Works in linear time
  - Generates compact functions (near space-optimal)
    - MPHF \((m = n)\): 3.3\(n\) bits
    - PHF \((m = 1.23n)\): 2.7\(n\) bits
    - Theoretical lower bound:
      - MPHF: 1.44\(n\) bits
      - PHF: 0.89\(n\) bits
Sequential External Perfect Hashing Algorithm

MPHF(x) = MPHF_i(x) + offset[i];
Key Set Does Not Fit In Internal Memory

Partitioning

Key Set S of $\beta$ bytes

$\mu$ bytes of Internal memory

File 1

$0 \leq k < 2^b - 1$

$N = \beta / \mu$  
$b = \text{Number of bits of each bucket address}$  
Each bucket $\leq 256$
Important Design Decisions

- We map long URLs to a fingerprint of fixed size using a hash function.
- Use our linear time and near space-optimal algorithm to generate the MPHF of each bucket.
- How do we obtain a linear time complexity?
  - Using internal radix sorting to form the buckets.
  - Using a heap of N entries to drive a N-way merge that reads the buckets from disk in one pass.
Algorithm Used for the Buckets:

Internal Random Access Memory Algorithm...
Internal Random Access Memory Algorithm

- Near space optimal
- Evaluation in constant time
- Function generation in linear time
- Simple to describe and implement
- Known algorithms with near-optimal space either:
  - Require exponential time for construction and evaluation, or
  - Use near-optimal space only asymptotically, for large $n$
- Acyclic random hypergraphs
  - Used before by Majewski et al. (1996): $O(n \log n)$ bits
- We proceed differently: $O(n)$ bits
  (we changed space complexity, close to theoretical lower bound)
Random Hypergraphs (r-graphs)

3-graph:

0  1

2  3

4  5

3-graph is induced by three uniform hash functions
Random Hypergraphs (r-graphs)

- 3-graph:

\[
\begin{align*}
    h_0(\text{jan}) &= 1 \\
    h_1(\text{jan}) &= 3 \\
    h_2(\text{jan}) &= 5
\end{align*}
\]

- 3-graph is induced by three uniform hash functions
Random Hypergraphs (r-graphs)

- 3-graph:

  \[
  h_0(\text{jan}) = 1 \quad h_1(\text{jan}) = 3 \quad h_2(\text{jan}) = 5
  \]
  \[
  h_0(\text{feb}) = 1 \quad h_1(\text{feb}) = 2 \quad h_2(\text{feb}) = 5
  \]

- 3-graph is induced by three uniform hash functions
Random Hypergraphs (r-graphs)

- 3-graph:
  - $h_0(\text{jan}) = 1 \quad h_1(\text{jan}) = 3 \quad h_2(\text{jan}) = 5$
  - $h_0(\text{feb}) = 1 \quad h_1(\text{feb}) = 2 \quad h_2(\text{feb}) = 5$
  - $h_0(\text{mar}) = 0 \quad h_1(\text{mar}) = 3 \quad h_2(\text{mar}) = 4$

- 3-graph is induced by three uniform hash functions
- Our best result uses 3-graphs
Acyclic 2-graph

\[ G_r: \]

0 \ 
\begin{array}{c}
\text{mar} \\
4 \\
\end{array} 
\begin{array}{c}
\text{jan} \\
5 \\
\end{array} 
\begin{array}{c}
\text{feb} \\
6 \\
\end{array} 
\begin{array}{c}
\text{apr} \\
7 \\
\end{array} 
\begin{array}{c}
3 \\
\h_0 \\
\end{array} 
\begin{array}{c}
\h_1 \\
\end{array}

L: \emptyset
Acyclic 2-graph

\[ G_r: \]

\[
\begin{array}{c c c c}
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
\end{array}
\]

\[ L: \{0, 5\} \]
Acyclic 2-graph

\[ G_r : \]

\begin{array}{cccc}
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
h_0 & & & h_1 \\
\end{array}

L: \{0,5\} \{2,6\}
Acyclic 2-graph

$G_r$:  

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
h_0 & h_1
\end{array}
\]

L: $\{0,5\} \{2,6\} \{2,7\}$
Acyclic 2-graph

$G_r$: 0 1 2 3 $h_0$

$G_r$ is acyclic

0 1 2 3

L: {0,5} {2,6} {2,7} {2,5}

4 5 6 7 $h_1$
Internal Random Access Memory Algorithm (r=2)
Internal Random Access Memory Algorithm ($r=2$)
Internal Random Access Memory Algorithm (r=2)

S

\[ \begin{align*}
\text{jan} & \quad \text{feb} & \quad \text{mar} & \quad \text{apr} \\
0 & \quad 1 & \quad 2 & \quad 3 \\
4 & \quad 5 & \quad 6 & \quad 7 \\
\end{align*} \]

Mapping

\[ G_r: \]

Assigning

\[ L: \{0,5\} \{2,6\} \{2,7\} \{2,5\} \]

\[ g \]

\[ \begin{align*}
0 & \quad r \\
1 & \quad r \\
2 & \quad r \\
3 & \quad r \\
4 & \quad r \\
5 & \quad r \\
6 & \quad r \\
7 & \quad r \\
\end{align*} \]
Internal Random Access Memory Algorithm (r=2)

Mapping

$S$

jan
feb
mar
apr

$G_r$:

0 1 2 3
mar jan feb apr

h_0
0 1 2 3

Assigning

L:

{0,5} {2,6} {2,7} {2,5}

$g$

0
1
2
3

0

4

5

6

7

r
r
r
r
r
r
r
Internal Random Access Memory Algorithm (r=2)

$S$:
- jan
- feb
- mar
- apr

$G_r$: 0 1 2 3 $h_0$
- 4
- 5
- 6
- 7 $h_1$

Mapping

Assigning

$L$: 
- 0: \{0,5\}
- 1: \{2,6\}
- 2: \{2,7\}
- 3: \{2,5\}

$g$
- 0: r
- 1: r
- 2: 0
- 3: r
- 4: r
- 5: r
- 6: r
- 7: 1
Internal Random Access Memory Algorithm (r=2)
Internal Random Access Memory Algorithm (r=2)

\[
i = (g[h_0(feb)] + g[h_1(feb)]) \mod r = (g[2] + g[6]) \mod 2 = 1
\]
Internal Random Access Memory Algorithm: PHF

\[ i = (g[h_0(\text{feb})] + g[h_1(\text{feb})]) \mod r = (g[2] + g[6]) \mod 2 = 1 \]

\[ \text{phf(feb)} = h_{i=1}(\text{feb}) = 6 \]
Internal Random Access Memory Algorithm: MPHF

\[ i = (g[h_0(feb)] + g[h_1(feb)]) \mod r = (g[2] + g[6]) \mod 2 = 1 \]

\[ \text{phf}(\text{feb}) = h_{i=1}(\text{feb}) = 6 \]

\[ \text{mphf}(\text{feb}) = \text{rank}(\text{phf}(\text{feb})) = \text{rank}(6) = 2 \]
Space to Represent the Function

Mapping

Assigning

2 bits for each entry
Space to Represent the Functions ($r = 3$)

- **PHF** $g: [0, m-1] \rightarrow \{0,1,2\}$
  - $m = cn$ bits, $c = 1.23 \rightarrow 2.46$ bits
  - $(\log 3) cn$ bits, $c = 1.23 \rightarrow 1.95 n$ bits (arith. coding)
  - Optimal: $0.89n$ bits

- **MPHF** $g: [0, m-1] \rightarrow \{0,1,2,3\}$ (ranking info required)
  - $2m + \varepsilon m = (2+ \varepsilon)cn$ bits
  - For $c = 1.23$ and $\varepsilon = 0.125 \rightarrow 2.62$ bits
  - Optimal: $1.44n$ bits.
Sufficient condition to work

Repeatedly selects $h_0, h_1..., h_{r-1}$

For $r = 3$, $m = 1.23n$: $\Pr_a$ tends to 1

Number of iterations is $1/\Pr_a = 1$
Experimental Results

- **Metrics:**
  - Generation time
  - Storage space for the description
  - Evaluation time

- **Collection:**
  - 1.024 billions of URLs collected from the web
  - 64 bytes long on average

- **Experiments**
  - Commodity PC with a cache of 4 Mbytes
  - 1.86 GHz, 1 GB, Linux, 64 bits architecture
### Generation Time of MPHFs (in Minutes)

<table>
<thead>
<tr>
<th>n (millions)</th>
<th>32</th>
<th>128</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential ECA</td>
<td>0.95 ± 0.02</td>
<td>5.1 ± 0.01</td>
<td>22.0 ± 0.13</td>
<td>46.2 ± 0.06</td>
</tr>
</tbody>
</table>
Related Algorithms

- Fox, Chen and Heath (1992) – FCH

All algorithms coded in the same framework
## Generation Time

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Generation Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal (r = 3)</td>
<td>6.7 ± 0.01</td>
</tr>
<tr>
<td>External</td>
<td>6.3 ± 0.11</td>
</tr>
<tr>
<td>MWHC</td>
<td>7.18 ± 0.01</td>
</tr>
<tr>
<td>FCH</td>
<td>2,400.1 ± 711.6</td>
</tr>
</tbody>
</table>

3,541,615 URLs
### Generation Time and Storage Space

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Generation Time (sec)</th>
<th>Space (bits/key)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal (r = 3)</td>
<td>6.7 ± 0.01</td>
<td>2.6</td>
</tr>
<tr>
<td>External</td>
<td>6.3 ± 0.11</td>
<td>3.1</td>
</tr>
<tr>
<td>MWHC</td>
<td>7.18 ± 0.01</td>
<td>26.76</td>
</tr>
<tr>
<td>FCH</td>
<td>2,400.1 ± 711.6</td>
<td>4.2</td>
</tr>
</tbody>
</table>

3,541,615 URLs
## Generation Time, Storage Space and Evaluation Time

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Generation Time (sec)</th>
<th>Space (bits/key)</th>
<th>Evaluation time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal (r = 3)</td>
<td>6.7 ± 0.01</td>
<td>2.6</td>
<td>2.1</td>
</tr>
<tr>
<td>External</td>
<td>6.3 ± 0.11</td>
<td>3.1</td>
<td>2.7</td>
</tr>
<tr>
<td>MWHC</td>
<td>7.18 ± 0.01</td>
<td>26.76</td>
<td>2.46</td>
</tr>
<tr>
<td>FCH</td>
<td>2,400.1 ± 711.6</td>
<td>4.2</td>
<td>1.6</td>
</tr>
</tbody>
</table>

3,541,615 URLs
Key length = 64 bytes
The Distributed External Memory Based Algorithm ...
Distributed Construction of the MPHFs

Manager

Worker 0

Worker 1

... Worker p-1
Distributed Construction of MPHFs

- **Manager:**
  - Assign tasks to workers
  - Determine global values during execution
  - Dump resulting MPHFs to disk

- **Worker:**
  - Has a partition of the keys on disk
  - Creates its buckets from the keys \((B_{pw} = N_b/p)\)
    - Migrate data whenever necessary
  - Constructs a MPHF for each bucket
Sequential External Perfect Hashing Algorithm

Worker 0  Worker 1  Worker p-1
0 1 2 3 4 5  

Key Set S

Partitioning

Worker 0  Worker p-1
h_0

Buckets

Searching

MPHF(x) = MPHF_i(x) + offset[i];

Hash Table
Partitioning Step in Each Worker

Diagram:
- Disk reader
- Disk
- Keys queue
Partitioning Step in Each Worker
Partitioning Step in Each Worker

Disk reader → Keys queue → Hashing → Fingerprint queue

Disk → Fingerprint queue

Sender → Fingerprint queue

Local network
Partitioning Step in Each Worker

- Disk reader
- Keys queue
- Hashing
- Fingerprint queue
- Net reader
- Local network
- Sender
- Disk
- Fingerprint queue
- Fingerprint queue
- Fingerprint queue
Partitioning Step in Each Worker

- Disk reader
- Disk
- Sender
- Local network
- Keys queue
- Hashing
- Fingerprint queue
- Block sorter
- Fingerprint queue
- Block queue
- Net reader
- Local network
Partitioning Step in Each Worker

Disk reader → Keys queue → Hashing → Fingerprint queue → Block sorter → Block queue → Block dumper

Disk reader → Net reader → Local network

Disk reader → Sender → Local network
Sequential External Perfect Hashing Algorithm

Worker 0

Worker 1

Worker p-1

0 1 2 3 4 5

... n-1

Key Set S

Partitioning

Worker 0

Worker p-1

h₀

Buckets

Worker 0

Worker 1

Worker p-1

MPHF₀ MPHF₁ MPHF₂

MPHF_{2^{b-1}}

0 1 2

2^{b-1} - 1

n-1

Hash Table

MPHF(x) = MPHF_i(x) + offset[i];
Searching Step in Each Worker
Searching Step in Each Worker

- Disk
  - Bucket reader
  - Buckets queue
    - MPHF gen 0
    - MPHF gen t-1
Description and Evaluation of a MPHF

- Centralized in one machine
- Distributed among the participating machines
Centralized Description and Evaluation of MPHFs

- End of the partitioning step:
  - Worker sends size of each bucket to manager
  - Manager calculates the offset array

- End of searching step (construction of MPHFs):
  - Worker sends MPHFs of its buckets to manager
  - Manager writes sequentially final MPHF to disk

- $MPHF(x) = MPHF_i(x) + offset[i]$
Distributed Description and Evaluation of MPHFs

- Description of MPHFs of a bucket
  - Stays in the bucket
- Evaluation of a MPHF
  - Locate the key inside the bucket:
    \[ MPHF_{\text{partial}}(k) = MPHF_i(k) + \text{localoffset}[i] \]
  - Add this to the number of keys before worker \( w \):
    \[ MPHF(k) = MPHF_{\text{partial}}(k) + \text{globaloffset}[w] \]
  - A key stream is evaluated in parallel
Advantages of Distributed Evaluation of MPHFs

- No need to send the MPHFs of a bucket to manager
- They are written to disk in parallel by the workers
- Final function is stored in a distributed way
  - Size of the description of the MPHF grows linearly with the size of the input key
On average, the number of keys $\tau$ sent through the net during the execution is:

$$\tau = \frac{n(p-1)}{p}$$

<table>
<thead>
<tr>
<th>$p$</th>
<th>Keys sent by a worker to the net</th>
<th>$\tau$ (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>75.008 74.994 75.000</td>
<td>75.000</td>
</tr>
<tr>
<td>10</td>
<td>90.009 89.991 90.000</td>
<td>90.000</td>
</tr>
<tr>
<td>14</td>
<td>92.864 92.849 92.857</td>
<td>92.857</td>
</tr>
</tbody>
</table>
Experimental Setup

- Three collections

<table>
<thead>
<tr>
<th>Collection</th>
<th>Avg. Key Size</th>
<th>$n$ (billions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>URLs</td>
<td>64</td>
<td>1.024</td>
</tr>
<tr>
<td>Random</td>
<td>16</td>
<td>1.024</td>
</tr>
<tr>
<td>Integers</td>
<td>8</td>
<td>1.024</td>
</tr>
</tbody>
</table>

- Cluster with 14 equal 64 bits single core machines
  - 2 gigabytes of main memory
  - 2.13 gigahertz
  - Linux operating system version 2.6
Speedup

- 64 bytes URLs

![Graph showing speedup versus number of machines for Linear, Centralized, and Distributed models. The graph plots speedup on the y-axis and number of machines on the x-axis, with a clear linear relationship for each model.]
Speedup

- 16 bytes random integers

![Graph showing speedup with respect to the number of machines. The graph includes lines for Linear, Centralized, and Distributed models, with speedup values on the y-axis and number of machines on the x-axis. The Centralized model shows the least speedup, followed by the Distributed model, and the Linear model shows the highest speedup.]
Speedup

- 8 bytes random integers

![Graph showing speedup with respect to number of machines]

- Linear
- Centralized
- Distributed
Scale-up

- 64 bytes URLs

![Graph showing scale-up comparison between Ideal, Centralized, and Distributed systems]
Scale-up

- 16 bytes random integers

![Graph showing scale-up with lines for Ideal scale-up, Centralized, and Distributed systems. The x-axis represents the number of machines, and the y-axis represents the scale-up factor. The line for Ideal scale-up remains constant, while Centralized and Distributed systems show an increasing trend with the number of machines.](image-url)
Scale-up

- 8 bytes random integers

![Graph showing scale-up with different lines for Ideal scale-up, Centralized, and Distributed systems. The x-axis represents the number of machines, and the y-axis represents scale-up.](image-url)
Scale-up

- 14.336 billion random integers
- 14 machines (each with 1.024 billion keys)

<table>
<thead>
<tr>
<th>n (billions)</th>
<th>Random Integer Collections</th>
<th>Construction time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Seq.</td>
</tr>
<tr>
<td>14.336</td>
<td>16-byte</td>
<td>41.17</td>
</tr>
<tr>
<td></td>
<td>8-byte</td>
<td>34.58</td>
</tr>
</tbody>
</table>
Load Balancing

- Execution time: fastest minus slowest (in minutes)

<table>
<thead>
<tr>
<th>p</th>
<th>$t_{fw}$</th>
<th>$t_{sw}$</th>
<th>$t_{sw} - t_{fw}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>12.78</td>
<td>12.86</td>
<td>0.09</td>
</tr>
<tr>
<td>10</td>
<td>4.32</td>
<td>4.40</td>
<td>0.07</td>
</tr>
<tr>
<td>14</td>
<td>3.76</td>
<td>3.84</td>
<td>0.08</td>
</tr>
</tbody>
</table>
Distributed Evaluation

- 1.024 billion key stream taken at random (minutes)

<table>
<thead>
<tr>
<th>Collection</th>
<th>Evaluation time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-byte URLs</td>
<td>33.1</td>
</tr>
<tr>
<td>16-byte Integers</td>
<td>24.5</td>
</tr>
<tr>
<td>8-byte Integers</td>
<td>18.2</td>
</tr>
</tbody>
</table>
C Minimal Perfect Hashing Library

- Why to build a library?
  - Lack of similar libraries in the free software community
  - Test the applicability of our algorithm out there

- Feedbacks:
  - 2,243 downloads (until May 27th, 2008)
  - Incorporated by Debian

- Library address: http://cmph.sourceforge.net
Conclusions

- Sequential and parallel perfect hashing algorithm
- Near space-optimal functions in linear time
- Function evaluation in time $O(1)$
- The algorithms are simpler and has much lower constant factors than existing theoretical results
- Outperforms the main practical general purpose algorithms found in the literature
Conclusions

- Construction time: 14 machines, 1 billion URLs
  - Sequential algorithm: 50 minutes
  - Parallel algorithm: 4 minutes
- Speedup > 90% for keys with more than 16 bytes
- Description and evaluation of MPHF:
  - Centralized
  - Distributed: fast evaluation for key streams