Contents

Hashing Algorithms

Simple Tabulation Hashing

Theory. Tabulation hashing is a hashing scheme which combines table lookups and `xor` operations in order to calculate the hash value. It views a key x as a vector of c characters $x_1, ..., x_c$, and relies on one totally random table T_i for each of the c character positions. The hash function simply performs a lookup for each character in the corresponding table, and `xor` the lookup-results together,

$$h(x) = T_1[x_1] \oplus, ..., \oplus T_c[x_c]$$

The tables are initialized prior to execution of the hashing function, which makes the complexity of the algorithm only depend on the speed of the table-lookups and the `xor`'ing (assumed *O(1)*). By constructing the table small enough to fit in fast cache, the table lookups very efficient, making algorithm is very fast while also being straightforward to implement.

If drawing keys from a universe of size u, we see that we have $O(u^{1/c})$ entries in each table, making the total amount of entries become $O(cu^{1/c})$. Since the c in the exponent outweights the factor c, the total space required for the tables is decreased as the amount of characters is increased. Thus, in order to ensure that the tables can fit in fast cache (optimally `L1`-cache), the size of the character could be decreased.

However, decreasing the size of the character yields more characters, which in turn yields more lookups and more `xor` operations, thus increasing the runtime of the algorithm. The decision of a good character-size is therefore a trade-off between having small enough characters for the tables to fit in fast cache, while not having too small characters to avoid too many computations.

The algorithm is by no means a new form of hashing, as the first instances of tabulashing was published in 1970 by Albert Zobrist [1], and was later rediscovered in greater generality by Carter & Wegman in 1979 [2]. The simple tabulation hashing is 3-independant, while better implementations of tabulation-based methods can supply 5-independance [3]. Pătraşcu and Thorup has recently shown that the low independance of the simple tabulation hashing can be shown to be non-fatal for many key applications [4], yielding strong hashing-properties for a very simple hashing method.

Implementation. The algorithm has been implemented in the `tabulation_hash.[h|cpp]`. It contains the `tabulation_hash` class, which holds the tables needed for the hashing, and only exposes one functions, namely `tabulation_hash::get_hash()`, which calculates the hash value based on the input string.

```
value_t tabulation_hash::get_hash(std::string key)
```

The type of the hash-values has been defined as `value_t`, which is set to `uint32_t`. By setting the character-size to 1B, there will be 256 entries in each table. Since each entry contains the hash-value for the given key, they will be of size 4B, making the total size of each table become |256*4B = 1kB|. Thus, to be able to hash a string of e.g. length 8, the memory needed to hold the tables will be 8kB. By allow character-sizes of 2B, amount of tables needed would be reduced by a factor of 2, but it would increase the amount of entries in each table by a factor 2^8, thus increasing the total size needed by a factor 2^7. Thus, for the same length 8 string, the memory required would become $2^{16} * 4 * (8/2)$B = 1MB, making the tables too large for fast cache. Therefore, the character-size has been fixed to 1B.

As for the generation of the random values in the tables, the suggestion of taking truly random values from `random.org` has been adhered to to some extend, as 16 tables of 256 entries each has been generated and hardcoded into the implementation. However, if more than 16 tables are needed, the tables are generated using a mersenne-twister in the constructor. This constructor takes as templating argument the maximum length of any key, which the hashing should be able to process, as this sets a limit to how many tables are needed, thus limiting the amount of space used.

The actual hashing works by splitting the key string into 8-bit chunks (i.e. `chars`), and using the 8 bits as an index in the table corresponding to the given chunk. The results of all these are simply `xor`'ed together.

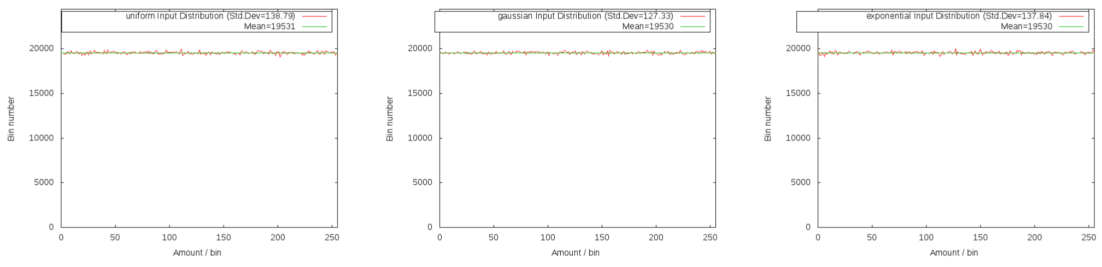
Experiments. To test the performance of the implementation, three kinds of experiments has been implemented:

- *Distribution Test*, where the distribution of many hash values is tested, for different input-distributions.
- *Key-Length Test*, where the run-time of the hashing algorithm is tested for various key lengths.
- *Multicore Test*, where the throughput of the hashing algorithm is tested for various amounts of cores.

Distribution Test. To test the distribution of the hashed output-values, three different input-distributions have been used:

- Uniform Distribution
- Gaussian Distribution
- Exponential Distribution

For each of the three distributions, 5.000.000 random 8-byte key-strings have been generated by generating a random integer using the given distribution, and interpreting it as a string. The keys have then been hashed, and the outputs have been categorized into 256 evenly sized bins. The distributions can been seen on Figure 1.



(a) Uniform Input Distr. (b) Gaussian Input Distr. (c) Exponential Input Distr.

Figure 1: Output Distributions of Tabulation Hashing

It can be seen, that for all three input distribution, the output distribution is very close to uniform, thus showing that tabulation hashing will map roughly the same amount of inputs to each hash value. Since the cost of hashing based methods increases drastically as more collisions occurs, a uniform output distribution will yield the best performance.

Key-Length Test. Testing of the speed of the algorithm on a single core is done by hashing keys of different length, and calculating the average runtime of each hashing. This test shows how well the implementation scales over key lengths.

To avoid filling the L1 cache with keys, we've chosen to only generate a small amount of strings (i.e. 5) for each of the lengths from 1-64 bytes. The strings are then hashed in groups of their lengths, from which the average run-time for each length can be calculated. To reduce the noise of the low amount of strings, the hashing of the groups are repeated a high amount of times. The average run-time for each length can be seen on figure 2. It contains four data-sets, representing hashing using 1, 8, 32 and 64 tabulation tables, respectively.

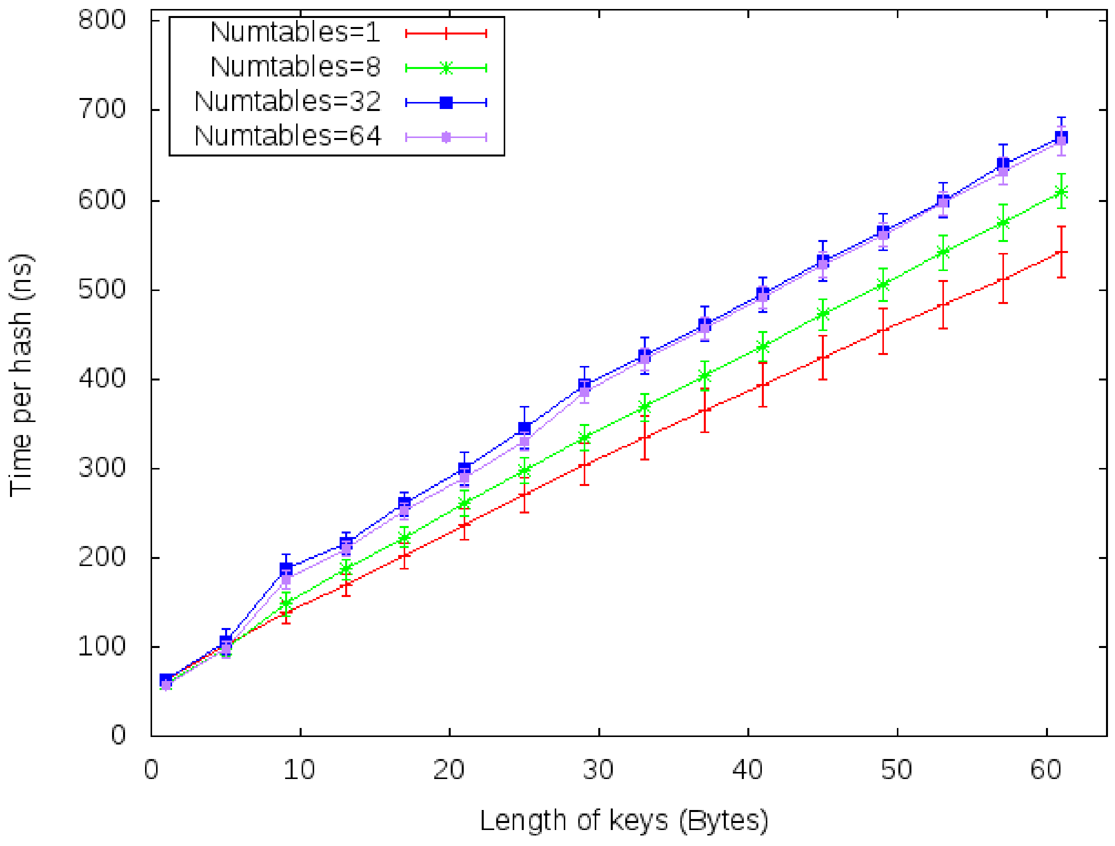


Figure 2: Run-times for different input string lengths.

It should first be noted that we see a close to linear increase in run-time as the length of the string is increased, in al of the four scenarios. In general, we've found that hashing a string of length 1 byte takes roughly 5ns, while each additional byte in the string adds half of a nanosecond. The low cost of the bytes following the first can be explained by that the reads of these bytes can be overlapped with the read of the previous bytes, where the first byte has to perform the entire read anyway.

Secondly, it should be noted that the four scenarios yields very similar results. Specifically, we can see that for short strings, the runtime is the same regardless of the amounts of tables. However, as the length of the strings is increased, we see that using a large amount of tables yields slower run-times than when using fewer tables. This is caused by the L1 cache being more contested by the high amount of tables and bytes in the strings.

Multicore Test To test the scalability across multiple cores, we've run a test similar to the key-length test on multiple threads, and measured the throughput of each thread, as well as the total througput. To do this, we've initialized one `tabulation_hash` object, which is used by all the threads in their computations. However, since the `tabulation_hash` object is never modified after initialization, a linear increase in throughput would be expected, as more cores are being utilized.

On Figure 3 the average throughput per thread, as well as the total throughput, is shown. Additionally, a linear function with slope equal to the throughput found when using just one thread has been added, which shows the optimal throughput scaling.

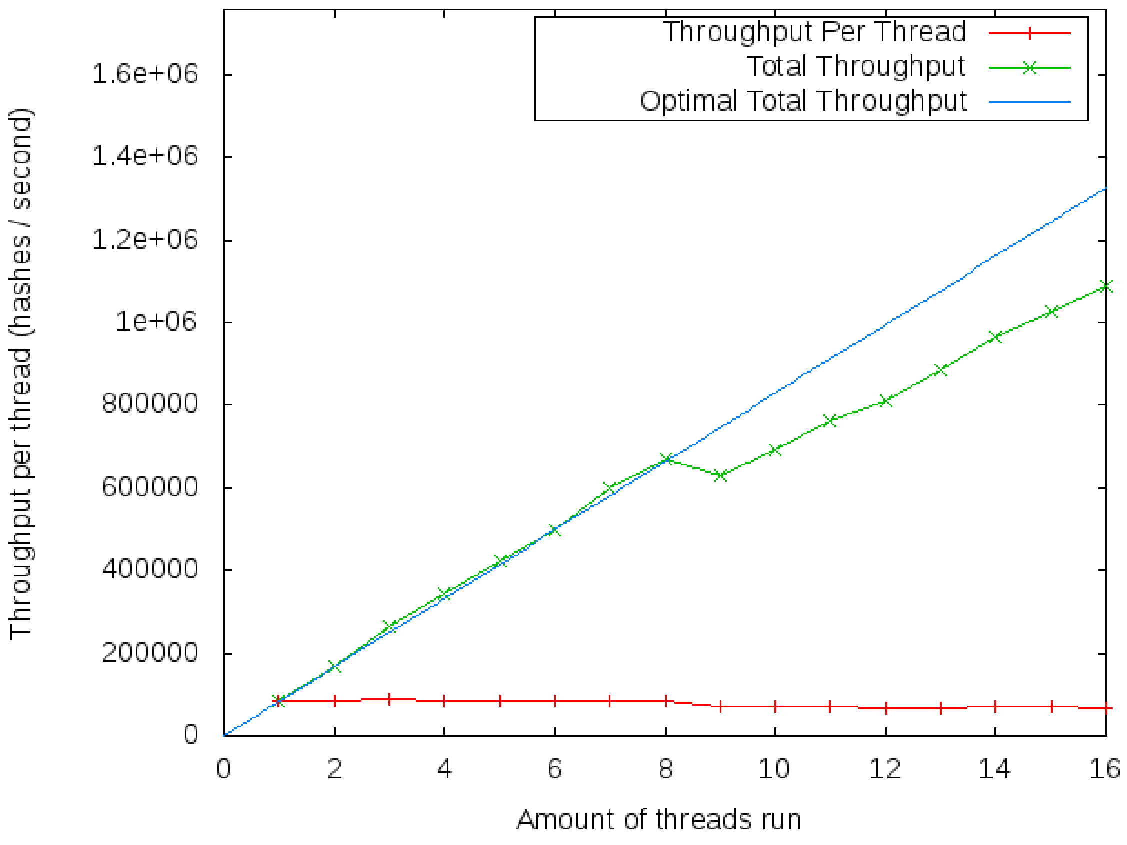


Figure 3: Throughput using different amounts of threads.

On the data-set for the average throughput per thread, we can see that for the first four cores, the throughput for each thread is almost as high as the throughput found when using just one thread. However, when running more threads than 8, the average throughput of each thread decreases. This can also be seen from the total throughput, which for the first 8 threads is close to the optimal throughput, while using more threads doesn't increase the total throughput. This could be explained by that the computer on which the tests has been run only has 8 physical cores.

Comparison to other hashing algorithms. Recently, a thorough comparison of hashing methods has been performed by Richter et al. [5], in which they compare four hash functions, namely

- Multiply-Shift Hashing

- Multiply-Add-Shift Hashing
- Tabulation Hashing
- Murmur Hashing

Since the first two only have implementations for fixed-length keys, we've chosen to focus on the comparison to murmur hashing. In order to compare the implementation of tabulation hashing with murmur hashing, an implementation of the most recent MurmurHash3 has been implemented for 32bit values.

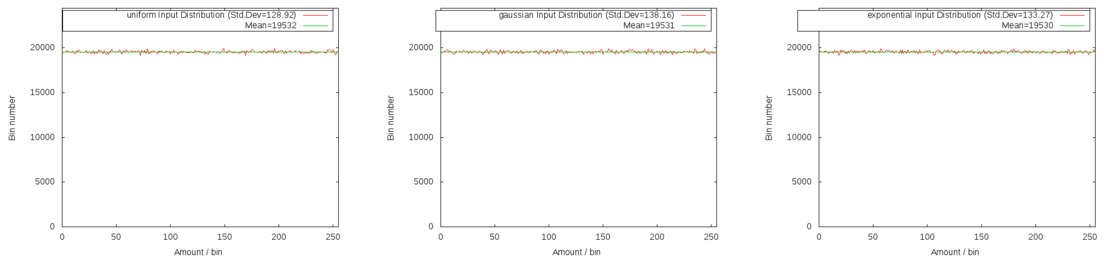
Murmur Hash.

Murmur hashing is a general purpose hash functoin, which is composed of several multiplications (MU) and rotations (R), thus creating the name (MU)(R)(MU)(R). The rotations are made by bit-shifts:

```
inline std::uint32_t rotl32 ( std::uint32_t x, std::int8_t r ) {
  return (x << r) | (x >> (32 - r));
}
```

This make the algorithm simple in terms of complexity, while yielding a good distrubition.

Distributions for murmur hash. As for the tabulation hash, we've evaluated the output distribution for murmur hash, when given the same three input distributions: Uniform, Gaussian and Exponential. The output distributions can be seen on the figure 4, where the distribution of 5.000.000 hash values can be seen, for each of the three cases.



(a) Uniform Input Distr. (b) Gaussian Input Distr. (c) Exponential Input Distr.

Figure 4: Output Distributions of Murmur Hashing

As seen, the output distributions of murmur hashing appears equally well distributed as the output distributions for tabulation hashing, which also if emphasized by noticing that the standard deviations of the two sets of output distributions are similar.

Hash-Based indexes

Extendible hashing

Theory. The extendible hashing data structure is based on the concept of a directory of pointers to buckets. The directory can be seen as a dynamically sized array, which from start-up is quite small, but which will grow when needed [1]. The buckets are containers for the data entries that contains a fixed amount of entries. To locate the correct bucket of a given hash-value, a `global_depth` is kept for the entire dictionary, which tells how many of the least significant bits of the hash value is required to find the bucket. The maximum amount of buckets in the dictionary is therefore always $2^{globaldepth-1}$.

When the directory has to grow (more on when this happens later), we simply double its size and create a copy of every bucket-pointer, and increment the `global_depth`. The newly included least significant bit is then used to distinguish between the original pointers and the newly copied pointers, where a 0 yields the old pointer, and a 1 yields the copy. Note that this maneuver does not create any new buckets, but simply allow for double as many buckets to be stored when needed. We do therefore end up with (at least) two pointers to each bucket, directly after the doubling of the directory.

Insertion. To insert an entry in the directory, the hash-value of the key is found, and the `global_depth` least significant bits are used to find the pointer to the actual bucket. If an empty slot is present in the given bucket, the value is simply inserted there, and the insertion is finished.

However, when an entry tries to enter a full bucket, additional space must be found. This is done by splitting the bucket into two, amongst which the entries in the original bucket (including the new entry to be inserted) are redistributed. To distinquish between the two buckets, an additional least significant bit is used, such that hash values, for which the additional bit is 0 remains in the old bucket, while hash values for which the additional bit is 1 is moved to the new bucket. To keep track of how many bits has to be used for each bucket, they contain an individual `local_depth`, which states how many least significant bits are needed, to uniquely distinquish the bucket. Thus, insertion into a full bucket `b` with `local_depth d` is done by the following pseudo-algorithm.

```
1 Create new empty bucket i, with same local_depth;
2 Insert i in directory, (1<<i.local_depth) places after b.
3 Increment b.local_depth and i.local_depth to d+1;
4 For each entry e in b:
  * Rehash e.key;
  * Insert e into the correct bucket based on the d+1 least significant bits;
5 Insert the new entry into correct bucket
6 Correct all pointers pointing to the original bucket;
```

Note the last step of the algorithm. If the `global_depth` is greater than the `local_depth`, there will be several pointers pointing to the original bucket. After splitting the buckets, these pointers should be changed to point to the correct of the two bucket, using all `local_depth` bits.

Also note that in step 2 of the algorithm, the directory might not be large enough to contain the place where the image bucket is to be inserted. This is the case if and only if the `global_depth` and `local_depth` of the given bucket (prior to incrementation) is equal, and then the directory is simply doubled (as described previously).

[1]Some implementations also allow directory to shrink under certain circumstances.

Removal. To remove an entry from the directory, the correct bucket of the entry is located, and if the entry is present, it is removed. Several modifications to the data can then be applied, in order to reduce the probing time and memory footprint.

The most simple change is to ensure that all entries in each bucket is always as early in the array as possible, thus not leaving any black entries in between used entries. This reduces the probing time, as an entry can be found no to be in a given bucket, as soon as a blank entry is found. Since the order of the entries in each bucket isn't important, this can be achieved by moving the last entry in the bucket to the place, where the removed entry previously was.

Implementation.

Concurrency.

Experiments.

References

[1] Zobrist, A. *A New Hasahing Method with Application for Game Playing.* http://research.cs.wisc.edu/techreports/1970/TR88.pdf

[2] Carter, J.; Wegman, Mark. *Universal classes of hash functions.* Journal of Computer and System Sciences, 143-154: http://ac.els-cdn.com/0022000079900448/1-s2.0-0022000079900448-main.pdf?_tid=f6ef3296-dbb5-11e5-996c-00000aacb35f&acdnat=1456401208_4e624f31178f38bc62b0b994c4b534b4

[3] Thorup, M.; Zhang, Y. *Tabulation based 5-universal hashing and linear probing.* In Proc. 12th Workshop on Algorithm Engineering and Experiments (ALENEX), 2009

[4] Pătraşcu, M.; Thorup, M. *The Power of Simple Tabulation Hashing* http://arxiv.org/abs/1011.5200

[5] Ritcher, S.; Alvarez, V.; Dittrich, J. *A seven Dimensional Analysis of Hashing Methods and its Implications on Query Processing* http://www.vldb.org/pvldb/vol9/p96-richter.pdf

[6] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009). *Introduction to Algorithms (3rd ed.).* Massachusetts Institute of Technology. pp. 253–280. ISBN 978-0-262-03384-8.