# PARALLEL IMPLEMENTATION OF HASHING ALGORITHMS

PROJECT REPORT FOR CS359

**Submitted by**

Kunal Gupta (150001015)

Keshav Goyal (150001014)

Computer Science and Engineering

3rd Year

*Under the Guidance of*

Dr. Surya Prakash


Department of Computer Science and Engineering

Indian Institute of Technology Indore

Autumn 2017

# CONTENTS

# INTRODUCTION

With increase in size and complexity of data, the storage and access of data is an important task. One way in which data is structured is in the forms of pairs, like Product-Price in an e-commerce business, Product-Stock in inventory systems, etc. Here the insertion, access and updating of the data can be very costly for large dataset size. This pair like association is generally implemented with hash tables/maps. Hashing is one of the methods used to map such key-value-like data.

In this project, we concentrate on the problem of implementing a parallel-friendly data structure that allows efficient random access of millions of elements and can be both constructed and accessed at interactive rates. Such a data structure has numerous applications in computer graphics, centered on applications that need to store a sparse set of items in a dense representation.

Cuckoo hashing is one of the schemes for resolving hash collisions of values of hash functions in a table, with worst-case constant lookup time. It is a so-called "multiple-choice" perfect hashing algorithm that achieves high occupancy at the cost of greater construction time.

# PROCEDURE

## SELECTION OF HASHING ALGORITHM

There are several algorithms that are used to hash values into a Hash Table. They may be divided on the basis of their way of handling collisions, some of which are:

- Linear Probing
  - Placing the new key into the closest following empty cell.
  - Results in primary clustering.
  - As the cluster grows larger, the search for those items hashing within the cluster becomes less efficient.
- Quadratic Probing
  - Given a collision with Hash Value *H*, the probing sequence now generated is:
    $$H + 1^2, H + 2^2, H + 3^2, H + 4^2, \dots, H + k^2$$
  - Provides good memory caching because it preserves some locality of reference.
  - There is no guarantee of finding an empty cell once the table gets more than half-full, or even before the table gets half-full if the table size is not prime.
- Cuckoo Hashing
  - Resolves collisions by using more than one (*k*) Hash Functions (generally, *k* = 2).
  - This provides more than one possible locations in the hash table for each key.

- Lookup requires inspection of just *k* locations in the hash table, which takes constant time in the worst case.
- However, *cycles* may be present.

We chose cuckoo hash as our open-addressing scheme to remove collisions as calculation of the next available location can be parallelized in a much simpler and efficient way.

## SELECTION OF BUCKETING AND HASHING FUNCTIONS

In our implementation, each key is assigned a bucket. Bucketing allows for collision reduction by restricting collisions to inside the bucket. Buckets may be assigned by a function such as:

$$b(k) = k \bmod |buckets|$$

$$or\ b(k) = [(c_0 + c_1 \cdot k) \bmod 1900813] \bmod |buckets|$$

$$where\ c_0\ and\ c_1\ are\ random\ integers\ and\ 1900813\ is\ a\ prime\ number.$$

We realized that for a bucket size $s$, such that $s \times |buckets| = Number\ of\ keys$, with a uniform distribution of keys in the range of 0 to $N$, the first hash function will suffice along with the buckets filling up uniformly. We choose $^{N}/_{409}$ buckets with bucket size as 409.

Now we use cuckoo hashing to hash values inside each bucket. Three hash functions will suffice, since using just two functions permits a load factor above 80% (theoretically). Algebraic hash functions of the following form are chosen:

$$g_i(k) = \left[\left(c_{i,0} + c_{i,1} \cdot k\right) \bmod 1900813\right] \bmod |T_i|$$

$$All\ constants\ c_{i,j}\ are\ random\ numbers.$$

By noticing the above functions, it can be seen that instead of having a single hash table, three hash tables, one for each hash function, are created to minimize collisions and allow for easier parallelization.

Initially we choose Table Size $T_i = 192$, thus overall load factor of $\frac{409}{192 \times 3} = 71\%$.

### IMPLEMENTATION

Initially we modified and appropriately designed the algorithm for sequential Cuckoo Hash and then the Parallel Cuckoo Hashing and then implemented the same in OpenMP.

### ANALYSIS

All analysis is done on an Intel i7-4710MQ Processor (4th Generation i7 with 4 cores/8 Threads).

- Speed-Up calculation for various input sizes, keeping all other parameters fixed.
- Number of collisions due to cycles for various table size, keeping input size fixed.
- Optimum table size to accommodate all keys by rebuilding the hash tables less than 100 times.

### DETECTING AND REMOVING PROBLEMS

As discussed further in the report.

## ALGORITHMS

### SEQUENTIAL CUCKOO HASH

The set of keys = $\boldsymbol{keys}$, hashing functions = $g_i(k)$ and Hash Tables = $T_i$ .

0. $flag = False$
1. $for\ \boldsymbol{k} \in \boldsymbol{keys}$:
2. $\quad loc_1 = g_1(k)$
3. $\quad if\ (k \notin T_1\ and\ k \notin T_2\ and\ k \notin T_3)$:
4. $\quad\quad T_1[loc_1] = k$
5. $for\ \boldsymbol{k} \in \boldsymbol{keys}$:
6. $\quad loc_2 = g_2(k)$
7. $\quad if\ (k \notin T_1\ and\ k \notin T_2\ and\ k \notin T_3)$:
8. $\quad\quad T_2[loc_2] = k$
9. $for\ \boldsymbol{k} \in \boldsymbol{keys}$:
10. $\quad loc_3 = g_3(k)$
11. $\quad if\ (k \notin T_1\ and\ k \notin T_2\ and\ k \notin T_3)$:
12. $\quad\quad T_3[loc_3] = k$
13. $for\ \boldsymbol{k} \in \boldsymbol{keys}$:
14. $\quad if\ (k \notin T_1\ and\ k \notin T_2\ and\ k \notin T_3)$:
15. $\quad\quad flag = True$
16. $if(flag)$:
17. $\quad goto\ Step\ 0$

## PARALLEL CUCKOO HASH

0. $flag = False$
1. $parfor\ \boldsymbol{k} \in \boldsymbol{keys}$:
2. $\quad loc_1 = g_1(k)$
3. $\quad if\ (k \notin T_1\ and\ k \notin T_2\ and\ k \notin T_3)$:
4. $\quad\quad lock(T_1[loc_1])$
5. $\quad\quad T_1[loc_1] = k$
6. $\quad\quad unlock(T_1[loc_1])$
7. $parfor\ \boldsymbol{k} \in \boldsymbol{keys}$:
8. $\quad loc_2 = g_2(k)$
9. $\quad if\ (k \notin T_1\ and\ k \notin T_2\ and\ k \notin T_3)$:
10. $\quad\quad lock(T_2[loc_2])$
11. $\quad\quad T_2[loc_2] = k$
12. $\quad\quad unlock(T_2[loc_2])$
13. $parfor\ \boldsymbol{k} \in \boldsymbol{keys}$:
14. $\quad loc_3 = g_3(k)$
15. $\quad if\ (k \notin T_1\ and\ k \notin T_2\ and\ k \notin T_3)$:
16. $\quad\quad lock(T_3[loc_3])$
17. $\quad\quad T_3[loc_3] = k$
18. $\quad\quad unlock(T_3[loc_3])$
19. $parfor\ \boldsymbol{k} \in \boldsymbol{keys}$:
20. $\quad if\ (k \notin T_1\ and\ k \notin T_2\ and\ k \notin T_3)$:
21. $\quad\quad flag = True$
22. $if(flag)$:
23. $\quad goto\ Step\ 0$

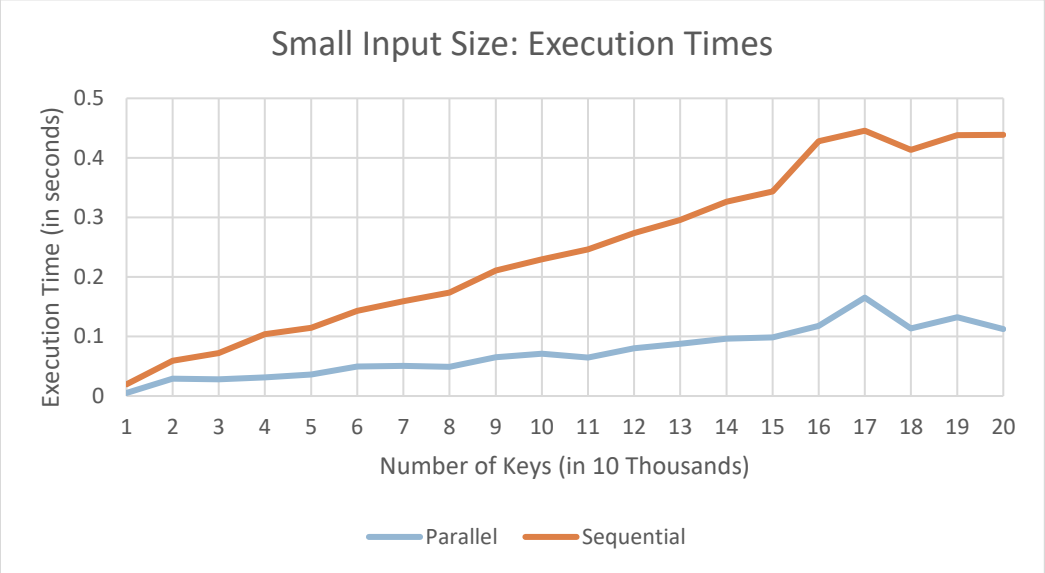
Here we follow the policy that the key, which collides to same value with same hash function and written earlier, is kicked out. Here, we have to lock the location before accessing; else, there will be corruption of data for obvious reasons.

# OBSERVATIONS AND ANALYSIS

## EFFECT OF INPUT SIZE ON EXECUTION TIME AND SPEEDUP

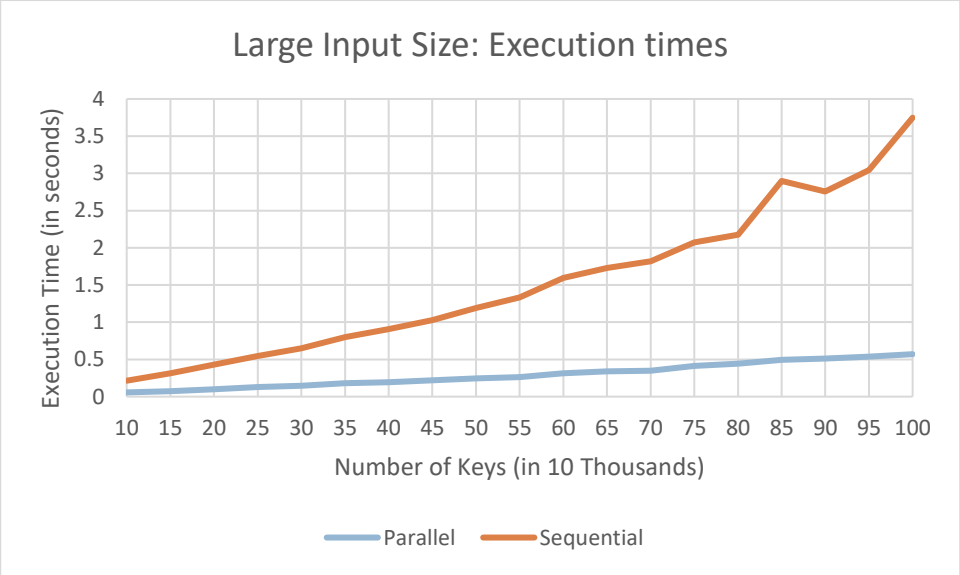
**SMALL INPUT**

| INPUT SIZE (IN 10 THOUSANDS) | SEQUENTIAL EXECUTION TIME (IN SECONDS) | PARALLEL EXECUTION TIME (IN SECONDS) | SPEED-UP |
|---|---|---|---|
| **1** | 0.014536 | 0.005092 | 2.854674 |
| **2** | 0.029888 | 0.029288 | 1.020486 |
| **3** | 0.043821 | 0.028131 | 1.557748 |
| **4** | 0.072915 | 0.031171 | 2.339193 |
| **5** | 0.078906 | 0.035846 | 2.20125 |
| **6** | 0.093383 | 0.049764 | 1.876517 |
| **7** | 0.108593 | 0.050698 | 2.141958 |
| **8** | 0.12498 | 0.048964 | 2.552488 |
| **9** | 0.145717 | 0.064958 | 2.243249 |
| **10** | 0.158111 | 0.071268 | 2.218541 |
| **11** | 0.181713 | 0.064454 | 2.819266 |
| **12** | 0.193379 | 0.080252 | 2.409647 |
| **13** | 0.207849 | 0.087912 | 2.364285 |
| **14** | 0.230006 | 0.0963 | 2.388432 |
| **15** | 0.245186 | 0.098222 | 2.496243 |
| **16** | 0.310394 | 0.117737 | 2.636334 |
| **17** | 0.280346 | 0.165282 | 1.696168 |
| **18** | 0.299863 | 0.113387 | 2.644598 |
| **19** | 0.306185 | 0.132157 | 2.316828 |
| **20** | 0.326376 | 0.112476 | 2.901739 |



## LARGE INPUT

| INPUT SIZE (IN 10 THOUSANDS) | SEQUENTIAL EXECUTION TIME (IN SECONDS) | PARALLEL EXECUTION TIME (IN SECONDS) | SPEED-UP |
|---|---|---|---|
| **10** | 0.157277 | 0.05715 | 2.752003 |
| **15** | 0.242187 | 0.07282 | 3.325831 |
| **20** | 0.331504 | 0.097005 | 3.417391 |
| **25** | 0.417652 | 0.129555 | 3.223743 |
| **30** | 0.504879 | 0.145762 | 3.463722 |
| **35** | 0.620299 | 0.179983 | 3.446431 |
| **40** | 0.712621 | 0.193404 | 3.684624 |
| **45** | 0.808007 | 0.22057 | 3.663268 |
| **50** | 0.949989 | 0.24317 | 3.906687 |
| **55** | 1.072394 | 0.261115 | 4.10698 |
| **60** | 1.280823 | 0.315909 | 4.054405 |
| **65** | 1.387326 | 0.340306 | 4.076702 |
| **70** | 1.468277 | 0.349124 | 4.205603 |
| **75** | 1.657462 | 0.413346 | 4.009866 |
| **80** | 1.732724 | 0.444641 | 3.896906 |
| **85** | 2.403379 | 0.495043 | 4.854889 |
| **90** | 2.244876 | 0.510905 | 4.393921 |
| **95** | 2.506213 | 0.537177 | 4.665526 |
| **100** | 3.177639 | 0.57041 | 5.570798 |



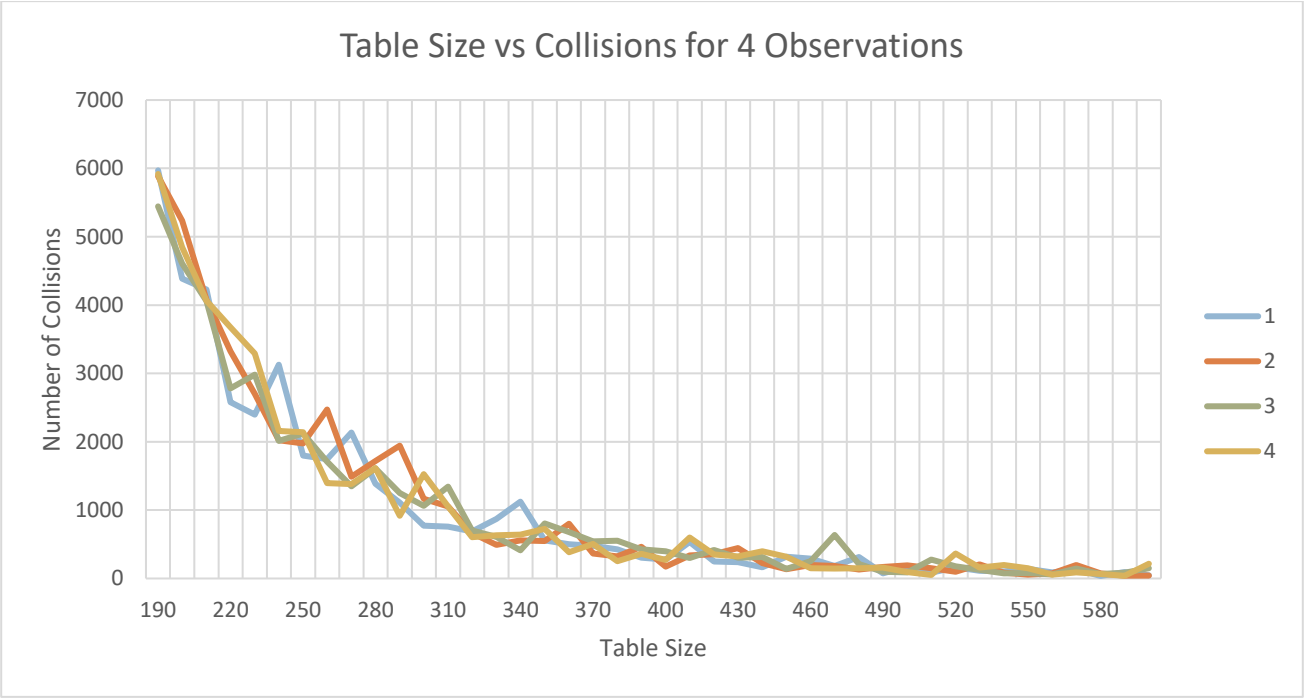

Large input size is thus better for performance analysis. We observe that speedup increases with the size of the input; and for a 10-fold increase in input size, sequential time increases by 20 times whereas, parallel time by only a 10-fold.

The execution time is linearly dependent on the input size as evident from the graph.

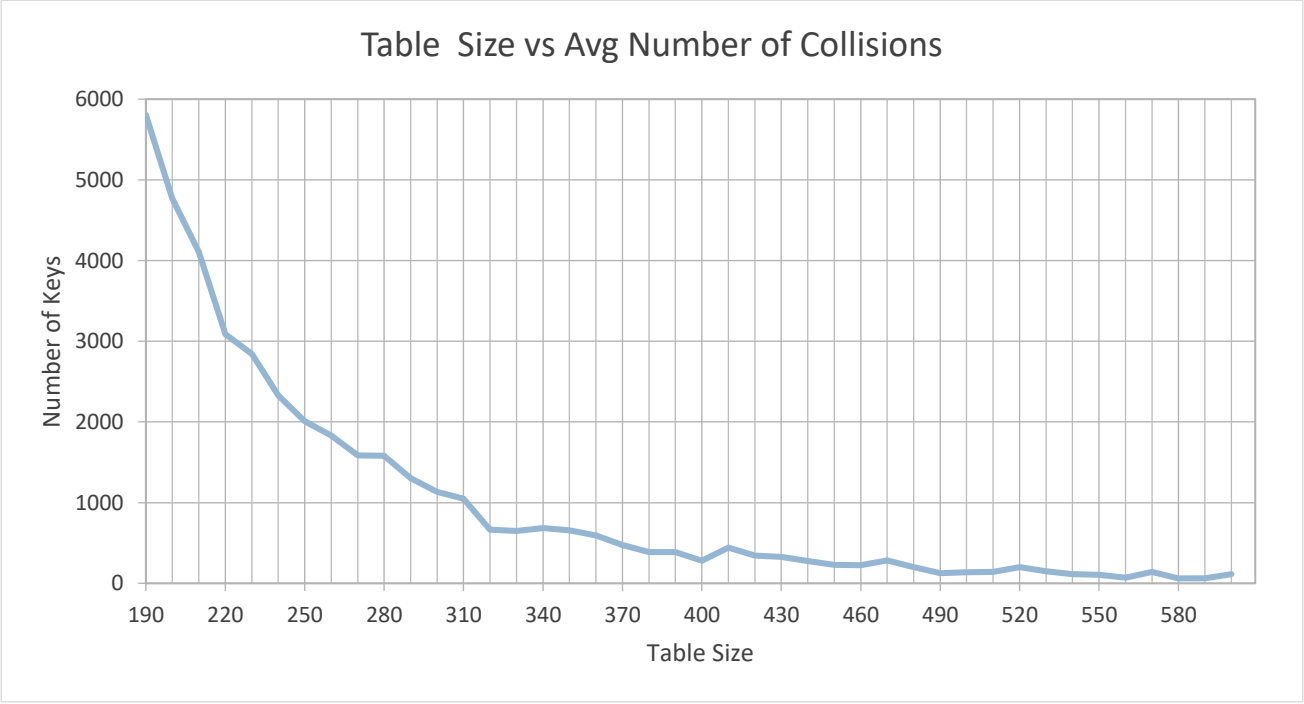
## EFFECT OF TABLE SIZE ON NUMBER OF COLLISIONS

Here we ran the algorithm for 100 iterations for each value of Hash Table Size and calculated average number of collisions for N=100,000. We repeated this 4 times:

| TABLE SIZE | AVERAGE NUMBER OF COLLISIONS | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| **190** | 5805 | 5971 | 5886 | 5444 | 5918 |
| **200** | 4769 | 4386 | 5234 | 4609 | 4848 |
| **210** | 4107 | 4228 | 4078 | 4057 | 4063 |
| **220** | 3089 | 2580 | 3322 | 2779 | 3675 |
| **230** | 2842 | 2397 | 2700 | 2982 | 3290 |
| **240** | 2331 | 3128 | 2024 | 2015 | 2157 |
| **250** | 2008 | 1799 | 1976 | 2120 | 2138 |
| **260** | 1830 | 1747 | 2470 | 1710 | 1394 |
| **270** | 1587 | 2135 | 1487 | 1346 | 1380 |
| **280** | 1583 | 1387 | 1718 | 1612 | 1616 |
| **290** | 1305 | 1110 | 1941 | 1251 | 917 |
| **300** | 1132 | 773 | 1167 | 1064 | 1525 |
| **310** | 1051 | 760 | 1051 | 1343 | 1051 |
| **320** | 666 | 688 | 667 | 703 | 606 |
| **330** | 651 | 871 | 490 | 610 | 631 |
| **340** | 683 | 1123 | 555 | 413 | 639 |
| **350** | 659 | 557 | 548 | 803 | 727 |
| **360** | 592 | 502 | 800 | 681 | 386 |
| **370** | 473 | 476 | 366 | 543 | 505 |
| **380** | 389 | 427 | 324 | 553 | 252 |
| **390** | 389 | 303 | 465 | 426 | 362 |
| **400** | 280 | 276 | 175 | 396 | 273 |
| **410** | 442 | 533 | 335 | 299 | 600 |
| **420** | 342 | 249 | 355 | 415 | 350 |
| **430** | 329 | 238 | 447 | 306 | 323 |
| **440** | 276 | 164 | 226 | 315 | 400 |
| **450** | 227 | 317 | 137 | 138 | 317 |
| **460** | 223 | 291 | 196 | 251 | 152 |
| **470** | 286 | 183 | 181 | 637 | 144 |
| **480** | 201 | 313 | 130 | 211 | 150 |
| **490** | 126 | 77 | 168 | 99 | 159 |
| **500** | 137 | 174 | 192 | 88 | 95 |
| **510** | 140 | 82 | 148 | 278 | 51 |
| **520** | 201 | 162 | 100 | 177 | 364 |
| **530** | 151 | 110 | 205 | 128 | 160 |
| **540** | 115 | 105 | 86 | 73 | 195 |
| **550** | 106 | 141 | 54 | 79 | 150 |
| **560** | 71 | 90 | 75 | 59 | 58 |
| **570** | 141 | 130 | 196 | 147 | 91 |
| **580** | 61 | 37 | 76 | 66 | 65 |
| **590** | 64 | 95 | 41 | 82 | 39 |
| **600** | 114 | 47 | 44 | 152 | 214 |




Since the trends in individual observations have many spikes, an average of the four observations will provide with better trends of the number of collisions with table size:




We observe that merely doubling the table size reduces the number of collisions to 0.16% of its initial value.

### OPTIMAL TABLE SIZE

From the last observations, we also noted the least table size that was required for the hash to be built in no more than 100 iterations of rebuilding (i.e. re-running cuckoo-hash) and selecting a new hash functions:

| OBSERVATION NUMBER | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| TABLE SIZE | 320 | 330 | 270 | 310 |

We take average of the least table size, $T_i \approx 307 \pm 22$ $(with\ load\ factor = 44\%)$ such that we always have a good chance of building the hash table in no more than 100 iterations of rebuilding the Hash Tables and regenerating new Hash Functions. This can further be checked for larger number of iterations of rebuilds to have a decreased table size and a load factor more than 44%.

## RESULTS

- The dependence of Execution Times on Input size seems to be linear.
- Speedup increased with increase in input size.
- The increase in the size of Hash Table led to a drastic decrease in the number of collisions.
- In order to get a good chance of accommodating the Hash Tables in less than 100 iterations of re-building the Hash Table, a good table size is $T_i = 307 \pm 22$ $(with\ load\ factor = 44\%)$

## CONCLUSIONS

Cuckoo Hash is a highly parallelizable algorithm reaching load factors of upto 90% (theoretical). For less than 100 rebuilds of the Hash Tables, a minimum of load factor = 44% for input of 100,000 keys.

The above results depict that its parallel algorithm outperforms the sequential algorithm by a considerable margin. The difference in performance is even larger when input size increases.

# DETECTING AND REMOVING PROBLEMS

## CYCLES

A cycle is kicking-out of elements in a cyclic fashion across hash tables. These cycles will lead to infinite loops. Detection of the cycle can be done by counting the number of times we had to perform kicks for all collision values and if it is more than a threshold value, we can assume that a cycle has been created. We can then regenerate the hash functions and start hashing from the scratch.

We tested the threshold to be equal to 25 and observed that increasing it did not lead to decreased number of collisions, thus it being a satisfactory value for determining a cycle.

## THEORETICAL TABLE SIZE

The theoretically optimum table size with a load factor of 71% did not work for small number of iterations of the algorithm, thus we looked at different table sizes and their efficiency of hashing in under 100 iterations, although there might be a dependence on the pseudo random generator of C++ (which we ignored).