Introduction:

The majority of parallel algorithms use the Parallel Random-Access Machine, or PRAM, concept. It facilitates the writing of a precursor parallel algorithm free from architecture restrictions and grants parallel-algorithm authors the luxury of treating processing capacity as limitless. It disregards how intricate inter-process communication is.

Assume for the moment that we want to add an array with N values. Generally, to find the array's sum, we loop across the array in steps of N. Let's assume that the array has a size of N and that each step takes one second. The iteration takes N seconds to finish as a result. Using a PRAM paradigm, which enables the processors to read from and write to the same memory region simultaneously, the same operation can be completed more quickly. The time required for execution is reduced if an array of size N has N/2 parallel processors.

A Cuckoo hash uses three tables to keep track of two or more distinct, identically sized tables that are each accessible via a distinct hash algorithm. In the event of a collision, keys are placed into the first table, removing any previously placed keys. One by one, the evicted keys are placed into the second table, the third table, and finally the first table again [5].

The procedure repeats itself until every key is inserted or until a maximum number of iterations is reached, at which point a construction failure is caused. When a process fails, it is resumed using an alternative hash function. Regretfully, at a limit load factor, the failure rate rises sharply given a number of tables [1]. This maximum is raised by using extra tables. At lower load factors, though, using an excessive number of tables becomes wasteful. In contrast, our system adjusts to these different scenarios on its own.

A Cuckoo hash is constructed in shared memory, a tiny but incredibly quick memory, by the parallel building process. First, a first level hash is used to randomly distribute the keys into buckets of equal size. A construction failure is triggered by any overflowing bucket. This restricts the maximum load factor that can be used; larger load factors result in an excessively high failure rate during this crucial distribution phase. Subsequently, Cuckoo hashing is paralleled with independent bucket hashing [4]. The newest hardware capabilities enable the single pass technique.

Example:

Aim :
To read a password list from a file, generate hashes out of them, and write the hashes to another file.

```
import hashlib
import multiprocessing
from concurrent.futures import ProcessPoolExecutor

def do_hash(x):
    return
hashlib.sha1(hashlib.sha1(x.encode()).digest()).hexdigest()

def hash_each_in_list(l):
    return [do_hash(x) for x in l]

def hash_each_in_list_parallel(l):
    n = multiprocessing.cpu_count()
    parts = [l[(i*len(l))//n:((i+1)*len(l))//n] for i in range(n)]
    with ProcessPoolExecutor() as executor:
        return sum(list(executor.map(hash_each_in_list, parts)), [])

l = hash_each_in_list_parallel(open('wordlists/pass.txt', 'r',
encoding='ISO-8859-1').read().splitlines())
with open('hashed.txt', 'a') as final:
    for f_result in l:
        final.write(f_result + '\n')
```

- The `hashlib` module is used to generate the SHA-1 hash of each password. In cryptography, SHA-1 (Secure Hash Algorithm 1) is a hash function that accepts an input and produces a 160-bit (20-byte) hash value known as a message digest (rendered as 40 hexadecimal digits) [3].
- The `multiprocessing` module is used to parallelize the hash generation process by supports spawning processes using an API [2].
- The `ProcessPoolExecutor` class is used to create a pool of worker processes.
- The `cpu_count()` function is used to determine the number of worker processes to create.
- The `map()` method of the `executor` object is used to apply the `hash_each_in_list()` function to each part of the password list in parallel.
- Finally, the `sum()` function is used to concatenate the results from each worker process into a single list.

Application:

Removing Duplicate Files Using Hashing and Parallel Processing in Python.

Aim:
Take path of the input folder which we need to remove duplicates. Once the program runs successfully, the output will be the same input folder without duplicates.

Approach:

1. Calculate the hash value for all files
2. Identify the Unique files using the hash values.
3. Delete the Duplicate Files.

First, we need to declare a function that takes file path as input. After reading the file, a function needs to be created which returns hash value for each file as an output. In some cases where we want to calculate hash value for only first few bytes of data rather than entire file, we can also send block size as parameter. In the code below, md5 hashing algorithm is being implemented.

```
def calculate_hash_val(path, block_size=''):
file = open(path, 'rb')
hasher = hashlib.md5()
data = file.read()
while len(data) > 0:
hasher.update(data)
data = file.read()
file.close()
return hasher.hexdigest()
```

Next, we need to declare a function that takes one empty dictionary and one dictionary with all input files as key and their respective hash value (calculated using the `calculate_hash_val` function) as their values. This function should return a unique dictionary `dic_unique` which contains no duplicates in the entire file.

```
def find_unique_files(dic_unique, dict1):
for key in dict1.keys():
if key not in dic_unique:
dic_unique[key] = dict1[key]
```

Lastly, once the unique files are identified, the final step is to delete the remaining duplicate files. We need to declare another function to delete the duplicates from input folder. The function takes two inputs `all_inps` and `unique_inps` which contains file paths and hash values respectively.

```
def remove_duplicate_files(all_inps ,unique_inps):
for file_name in all_inps.keys():
```

```
if all_inps[file_name] in unique_inps and
file_name!=unique_inps[all_inps[file_name]]:
os.remove(file_name)
elif all_inps[file_name] not in unique_inps:
os.remove(file_name)


if __name__ == '__main__':
pool = multiprocessing.Pool()
pool = multiprocessing.Pool(processes=4)
keys_dict = pool.map(calculate_hash_val, input_files)
pool.close()

inp_dups = dict(zip(keys_dict, input_files))
all_inps = dict(zip(input_files, keys_dict))

find_unique_files(unique_inps, inp_dups)
remove_duplicate_files(all_inps, unique_inps)
```

Conclusion:

Lessons learned:

- Parallel hashing can significantly reduce the time required to hash large amounts of data by processing multiple chunks simultaneously. This is particularly advantageous when dealing with large datasets or when performing hash computations in real-time applications.
- Utilizing multiple processing units or cores allows for better resource utilization.
- As the volume of data increases, parallel hashing provides a scalable solution by distributing the workload across multiple processors.
- Parallel hashing can be implemented with load balancing mechanisms to ensure that the workload is evenly distributed among processing units.

References:

[1] Karlin, Anna & Upfal,Eli. (1988). Parallel Hashing: An Efficient Implementation of Shared Memory.

[2] https://docs.python.org/3/library/multiprocessing.html

[3] Sahu, Aradhana & Ghosh, Samarendra. (2017). Review Paper on Secure Hash Algorithm With Its Variants. 10.13140/RG.2.2.13855.05289.

[4] Ismael Garc´ıa, Sylvain Lefebvre, Samuel Hornus, Anass Lasram. (2011). Coherent Parallel Hashing. Vol. 30, No. 6, Article 161

[5] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, Nina Amenta. (2009). Real-Time Parallel Hashing on the GPU. Vol. 28, No. 5, Article 154