Password decryption with Java Thread

Giulio Bazzanti Niccolò Biondi

Parallel Computing, April 2020

Table of Contents

1. Password Decryption
2. Sequential approach
3. Parallel approach
4. Experiments
5. Conclusions

Password Decryption

DES and brute-force attack

Data Encryption Standard (DES)

- is a block cipher algorithm
- is composed by sixteen phases
- adopts a secret key for both crypt and decrypt phase

DES and brute-force attack

Data Encryption Standard (DES)

- is a block cipher algorithm
- is composed by sixteen phases
- adopts a secret key for both crypt and decrypt phase

A brute-force attack

- is used to decrypt any encrypted passwords
- leverages all possible combinations inside the set [a-zA-Z0-9]
- wants to find a secret password

DES and brute-force attack

Data Encryption Standard (DES)

- is a block cipher algorithm
- is composed by sixteen phases
- adopts a secret key for both crypt and decrypt phase

A brute-force attack

- is used to decrypt any encrypted passwords
- leverages all possible combinations inside the set [a-zA-Z0-9]
- wants to find a secret password

A lot of combinations to analyze (with 8-characters passwords are 68^8)

Dictionary attack and Java Thread

Brute-force attack ⇒ **Dictionary Attack**

- use dictionary
  - less password to analyze
  - pre-computed 8-characters passwords

Dictionary attack and Java Thread

Brute-force attack ⇒ **Dictionary Attack**

- use dictionary
  - less password to analyze
  - pre-computed 8-characters passwords

- use Java Threads parallelism
  - different threads analyze different dictionary chunks
  - speed up the password analysis process

Need threads synchronization ⇒ three approaches

Sequential approach

Sequential approach

- The hacker knows the DES SecretKey
- Passwords analyzed sequentially

Sequential approach

- The hacker knows the DES SecretKey
- Passwords analyzed sequentially

Algorithm 1 Dictionary finder

```
Read the dictionary
for password in dictionary do
    Encrypt password $p$ to obtain $ep$
    Compare $ep$ with the target password
    if $ep$ is equal to target password then
        exit and return password
    end if
end for
```

Sequential approach

Very **expensive** task

- A single process checks all passwords in the dictionary
- Total computational time ⇒ $O(l)$

Typically the dictionary size is 10 GB

Parallel approach

Java Thread

Assumptions

- Multiple threads check different dictionary chunks
- **Callable** vs **Runnable**

Java Thread

Assumptions

- Multiple threads check different dictionary chunks
- **Callable** vs **Runnable**

Synchronization methods

- Threads Synchronization
- ReadWriteLock
- Atomic

Threads Synchronization

- Control access of multiple threads to shared resources
- Solve consistency problem and prevent race conditions
- The keyword *synchronized* leverages the lock built in Java Object

Threads Synchronization

- Control access of multiple threads to shared resources
- Solve consistency problem and prevent race conditions
- The keyword *synchronized* leverages the lock built in Java Object

State class

Two different functions to access to the shared variable

- *getState()*: read the boolean variable
- *setState()*: set the boolean state

ReadWriteLock

- Gives an advanced thread lock mechanism
- **ReentrantReadWriteLock** allows multiple readings but single writing
- Two different lock, one for reading and one for writing

ReadWriteLock

- Gives an advanced thread lock mechanism
- **ReentrantReadWriteLock** allows multiple readings but single writing
- Two different lock, one for reading and one for writing

State class

Two different functions to access to the shared variable

- *getState()*: read the boolean variable
- *setState()*: set the boolean state

Atomic

- Allows atomic access and updates without synchronization or locks
- Can implement non-blocking, lock-free algorithms solving race conditions
- No need a *State* class, only an Atomic boolean

Atomic

- Allows atomic access and updates without synchronization or locks
- Can implement non-blocking, lock-free algorithms solving race conditions
- No need a *State* class, only an Atomic boolean

AtomicBoolean

Two functions to interact with shared resources

- *get()*: return the value from the memory
- *set()*: write the value to memory

Callable vs Runnable

- Callable return a value, Runnable no
- Callable uses **Future** to store the return value

Callable vs Runnable

- Callable return a value, Runnable no
- Callable uses **Future** to store the return value
- Both leverage an **Executor**
  - Creates threads from a pool
  - Does not create extra threads
  - Separates threads creation and management

Experiments

Testing Hypothesis

- The hacker uses two preset dictionaries
  - *Dictionary_words* contains 19958400 8 letter English words.
  - *Dictionary_data* contains 461313 data from 1600 to 2020
- Two different experiments:
  - The hacker has to find different passwords
  - The hacker can use different threads number
- 10 runs for each testing configuration

Evaluation Metrics

Speedup

- Measures two approaches performance
- Use sequential and parallel time
- Depend on the CPU processor number

$$S_p = t_s / t_p$$

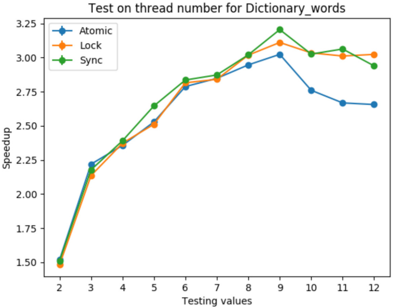
Efficiency

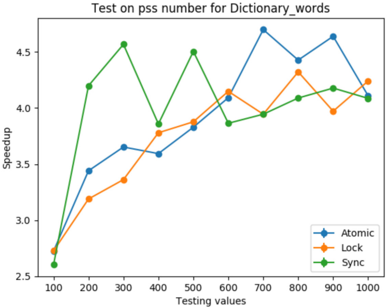
- Measures how well-utilized the processors
- Use the Speedup values
- Depend on the CPU processor number

$$E_p = S_p / p.$$

Dictionary_words

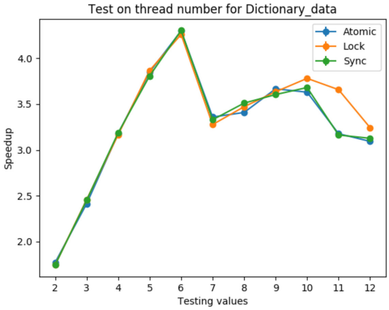
- More threads means more (smaller) dictionary chunks
- More passwords to retrieve increasing Speedup

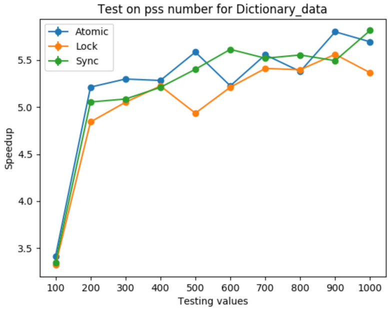

Test on thread number for Dictionary_words


Test on pss number for Dictionary_words

Dictionary_data

- Similar trend but higher performances
- Smaller dictionary ⇒ higher Speedup
- Password position in chuck affects the result



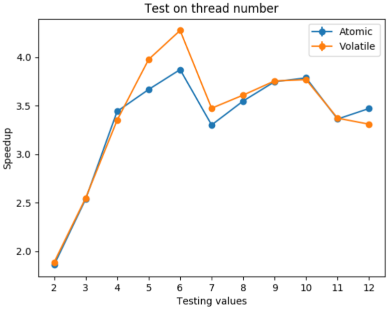


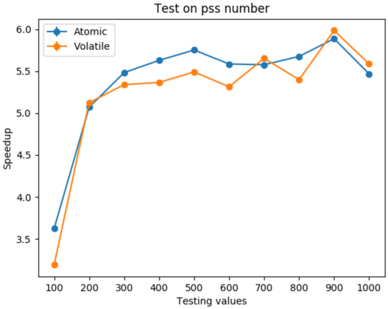
Atomic vs Volatile

- Volatile variable has atomic reading/writing operations
- One thread updates the shared variable ⇒ few synchronizations

Atomic vs Volatile

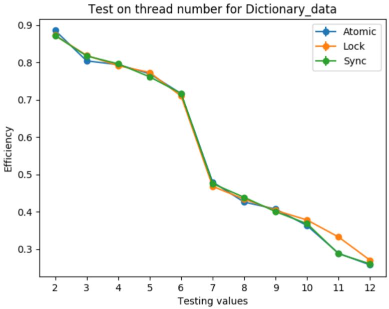
- Volatile variable has atomic reading/writing operations
- One thread updates the shared variable ⇒ few synchronizations
- Similar Speedup values w.r.t. Atomic ones

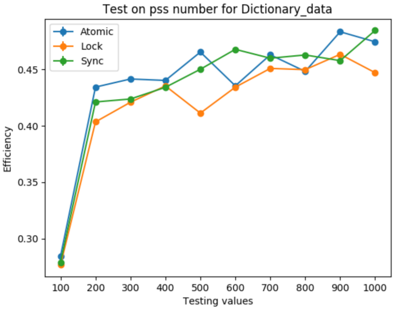




Efficiency

- Fixed passwords number and more thread ⇒ less thread work
- More passwords to retrieve ⇒ more passwords in each chunk


Test on thread number for Dictionary_data


Test on pss number for Dictionary_data

Conclusions

Conclusions

- Parallel approaches improves the computational time
- Bigger dictionary size implies more computational time
- The password position in dictionary affects the results
- Different synchronization methods not change Speedup
- No race conditions, few synchronization $\Rightarrow$ *Volatile*

Thanks for the attention