**Merkle root Calculation Documentation**

**Algorithm 1**

+++++++++

numRoutines := goroutines

numLeaves := len(m.Leaves)

m.Depth = calTreeDepth(numLeaves)

m.tree = make([][][]byte, m.Depth)

m.tree[0] = make([][]byte, numLeaves)

copy(m.tree[0], m.Leaves)

var prevLen int

m.tree[0], prevLen, err = m.fixOdd(m.tree[0], numLeaves)

if err != nil {

return

}

for i := uint32(0); i < m.Depth-1; i++ {

m.tree[i+1] = make([][]byte, prevLen>>1)

g := new(errgroup.Group)

for j := 0; j < numRoutines && j < prevLen; j++ {

idx := j << 1

g.Go(func() error {

for k := idx; k < prevLen; k += numRoutines << 1 {

newHash, err := m.HashFunc(append(m.tree[i][k], m.tree[i][k+1]...))

if err != nil {

return err

}

m.tree[i+1][k>>1] = newHash

}

return nil

})

}

if err = g.Wait(); err != nil {

return

}

m.tree[i+1], prevLen, err = m.fixOdd(m.tree[i+1], len(m.tree[i+1]))

if err != nil {

return

}

}

m.Root, err = m.HashFunc(append(m.tree[m.Depth-1][0], m.tree[m.Depth-1][1]...))

//fmt.Print((m.Root))

//fmt.Print("\n")

if err != nil {

return err

}

//wg.Wait()



Merkle tree level 3



Merkle tree level 2 



Merkle tree level 1



Merkle tree level 0

0 1 2 3 4 5 6 7

Sample Algorithm run

Build level tree using 2 Number of go routines

Merkle tree level 0 contains 8 leaves.

Merkle tree level 1 -

First go routine will process

1)Leaves 0 , 1 (concat)

2)Leaves 4 (0+4) , 5 (concat)

Loop which runs in a go routine increases by value of number of goroutines \* 2

so 0 and 4 th index and consecutive indexes will be processed in go routine 1

Second go routine will process

1)Leaves 2, 3 (concat)

2)Leaves 6 (2+4),7 (concat)

Loop which runs in a go routine increases by value of number of goroutines \* 2

so 2 and 6 th index and consecutive indexes will be processed in go routine 2

Merkle tree level 2 – (Number of nodes are halved)

First go routine will process

1)Nodes 0,1 (concat)

0 + 4 > final node index (loop terminates for go routine 1)

Second go routine will process

2) Nodes 2,3 (concat)

1+ 5 > final node index (loop terminates for go routine 2)

Merkle tree level 3

finally first go routine will process Nodes 0, 1 and compute root hash

g.Wait() at the end of one level loop prcoess ensures, next depth is not prcoessed till previous depth is finished.

**Algorithm 2**

++++++++

func ConcatHash(left, right []byte) []byte {

merged := make([]byte, 0, len(left)+len(right))

var err error

merged, err = defaultHashFunc(append(left, right...))

if err != nil {

log.Print(err)

}

return merged

}

func RootCalculator(data [][]byte) []byte {

if len(data) <= 1 {

return data[0]

}

done := make(chan bool)

mid := len(data) / 2

var left []byte

go func() {

left = RootCalculator(data[:mid])

done <- true

}()

right := RootCalculator(data[mid:])

<-done

return ConcatHash(left, right)

}

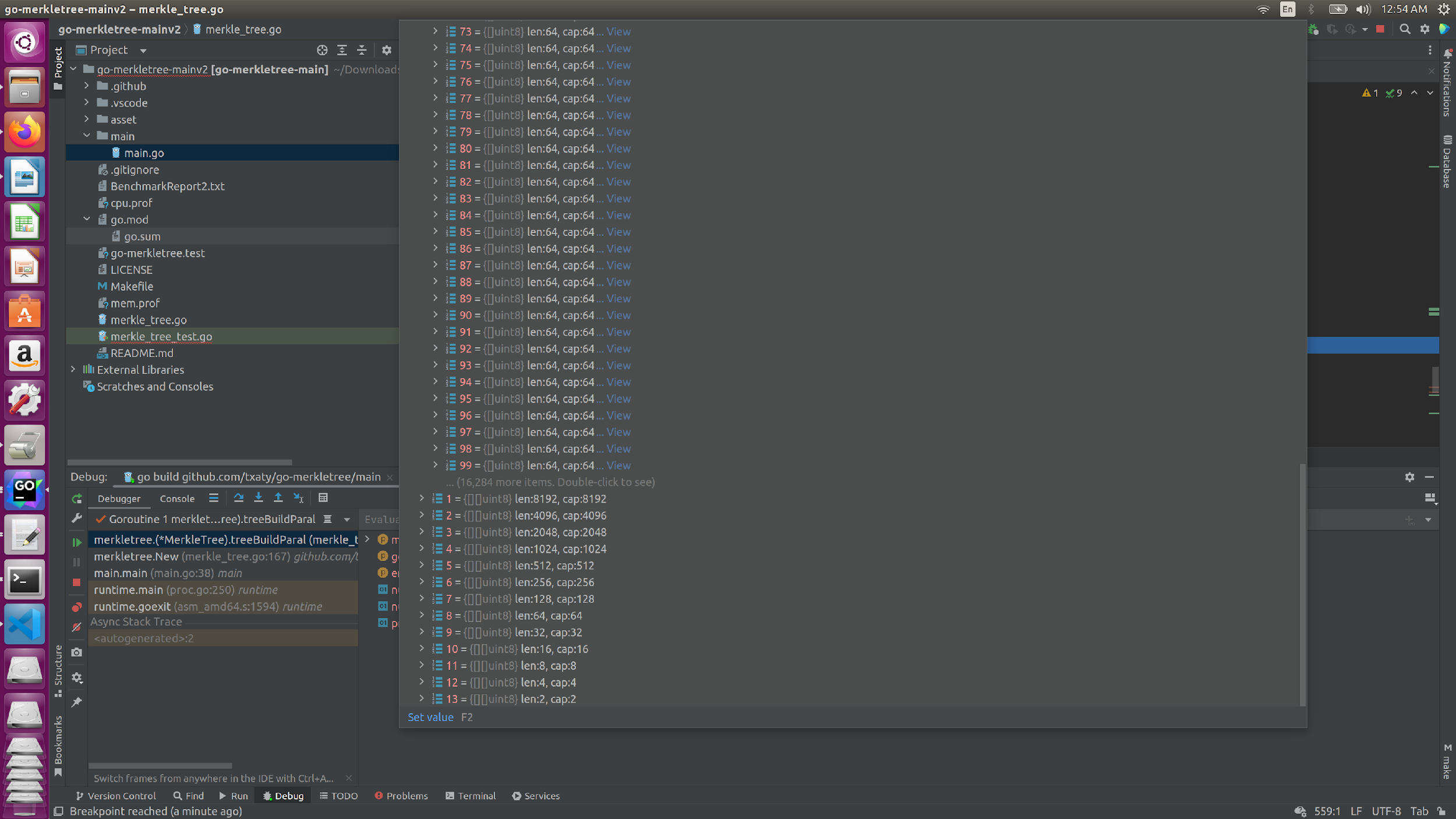
Firstly we keep dividing our merke leaves array recursively into the right side and the left side and call the Root calculation function on both sides in bottom of root calculation. Both these two calls can be made independently and we can execute one of them in a goroutine.

Now we have to make sure that ConcatHash(left,right) is executed only once we get the return values from both the recursive calls, i.e. both the left and right have been updated before ConcastHash(left,right) has to execute. Hence, we introduce a channel of type bool and send true on it as soon as left = Rootcalculator(data[:mid]) is executed. The <-done operation blocks the code before the statement ConcatHash(left,right) so that it does not proceed until our goroutine has finished. After the goroutine has finished and we receive true on the done channel, the code proceeds forward to ConcatHash(left,right).

Divide algorithm into independent chunks of code to execute using goroutines and synchronize the concurrent operations using channels.

**Merkle tree view**

**Leaves and depth view**



**Merkle tree Proof generation for a given leaf**

Merkle tree level 3 



Merkle tree level 2 



Merkle tree level 1



Merkle tree level 0

0 1 2 3 4 5 6 7

**Sample Algorithm run**

Data block is leaves with index 3.

for depth = 0

If leaf index is odd index sibling is even.

sibling = leaf index (3-1) 2

idx is divided by 2 for next depth

3/2

idx =1

for depth = 1

if leaf index is odd sibling is even.

Sibling = leaf index (1-1) 0

idx = 0/2

0

for depth = 2

if leaf index is even i.e 0, sibling is odd

Sibling = leaf index (0+1) 1

Siblings for leaf index 3 – depth 0 (leaf 2) , depth 1 (node 0), depth 2 (node 1)

Using siblings and given leaf index, root hash can be computed and verified given data block is correct.

For proof verification of a data block, its siblings are stored in proof structure.

**Faster algorithm for faster merkle tree root hash calculations**

1)"crypto/sha256"

2)fastsha256 "github.com/minio/sha256-simd"

Accelerate SHA256 computations in pure Go using AVX512, SHA Extensions for x86 and ARM64 for ARM. On AVX512 it provides an up to 8x improvement (over 3 GB/s per core). SHA Extensions give a performance boost of close to 4x over native.

This helps up in speeding up root hash calculation as hash concat function takes less time.

3)blake 3