A Tale of Performant Breadth First Search

the process of optimizing code

Egon Elbre

Golang Estonia Meetup @ Mooncascade

2020-02-04

Disclaimer

I won't explain all the details.

If you don't get them, it's fine.

Don’t worry about exact numbers.

Follow the overall process instead.

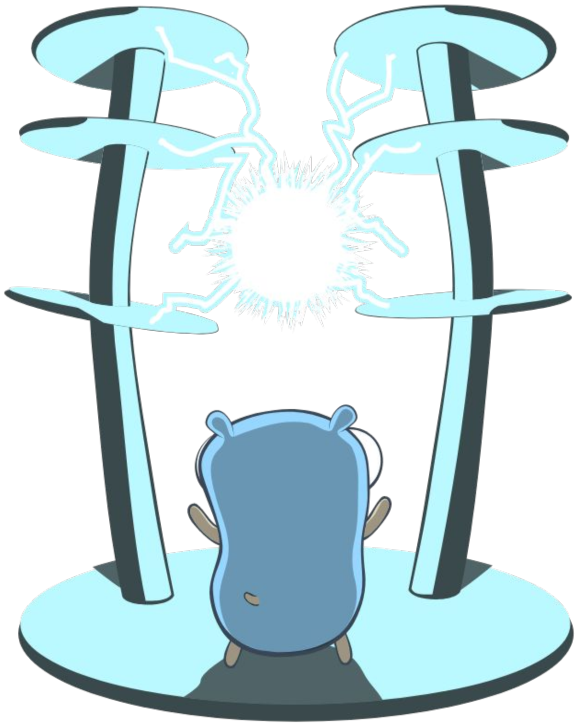
Disclaimer
Part 2

95%* of your code doesn't need this.

** there are exceptions*

Basic Process

1. Measure
2. Make a few hypotheses
3. Formulate few potential fixes
4. Try all promising solutions (and measure)
5. Keep the best solution (if there is one)
6. Keep the second best solution around
7. Repeat



Knowledge helps, but is not required

1. Measure

2. Make a few hypotheses

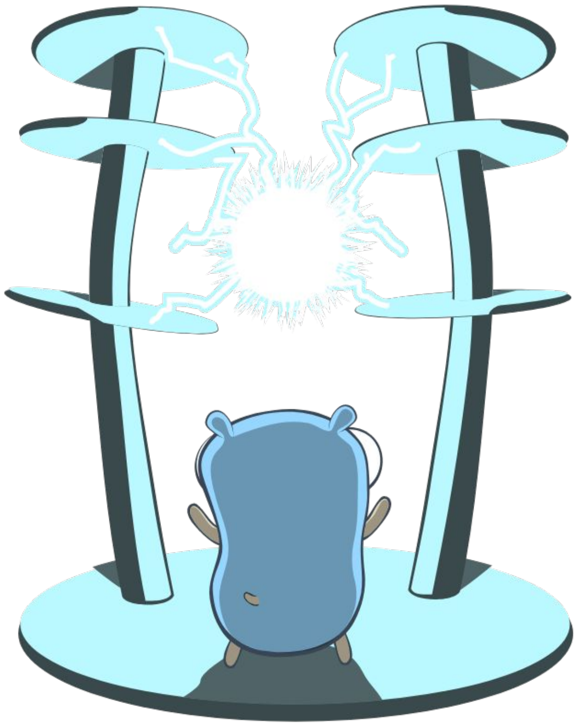
3. Formulate few potential fixes

4. Try all promising solutions (and measure)

5. Keep the best solution (if there is one)

6. Keep the second best solution around

7. Repeat



Knowledge helps

#performance in Gophers Slack

The Algorithm Design Manual
by Steven Skiena

https://github.com/dgryski/go-perfbook
by Damian Gryski

Basic ideas for modern processors

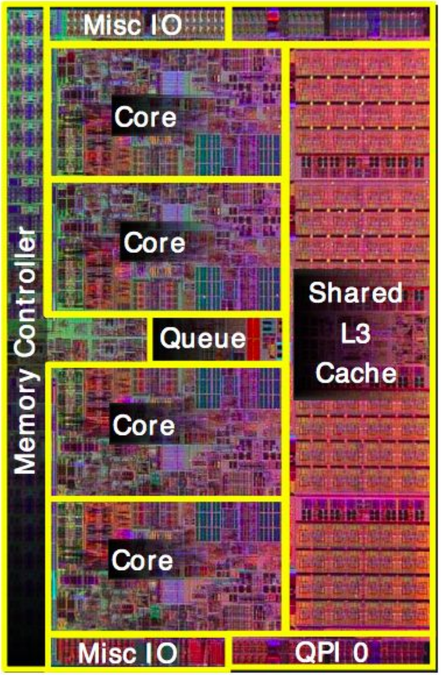
Use better algorithms.

Make it predictable.

Use less memory.

Use fewer pointers.

Keep computations close.



https://commons.wikimedia.org/wiki/File:Shared_cache_cpu.png

Breadth First Search

Backstory

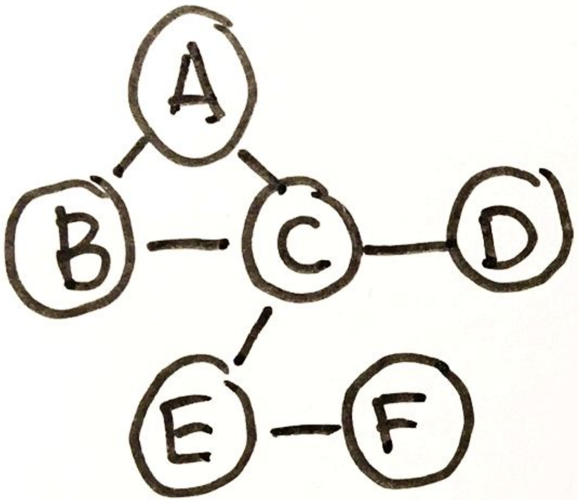
https://gophers.slack.com/archives/C0VP8EF3R/p1520745959000037

Dataset - Friendster social network

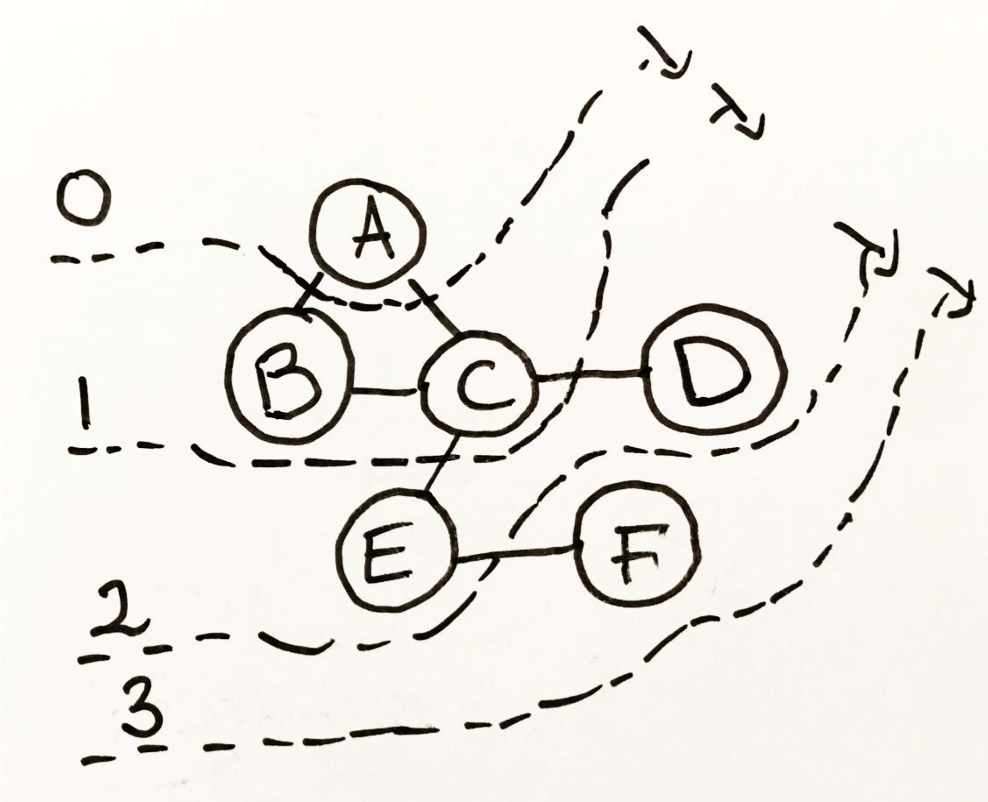
Dataset statistics	
Nodes	65608366
Edges	1806067135
Nodes in largest WCC	65608366 (1.000)
Edges in largest WCC	1806067135 (1.000)
Nodes in largest SCC	65608366 (1.000)
Edges in largest SCC	1806067135 (1.000)
Average clustering coefficient	0.1623
Number of triangles	4173724142
Fraction of closed triangles	0.005859
Diameter (longest shortest path)	32

65 million nodes
1.8 billion edges

Graph



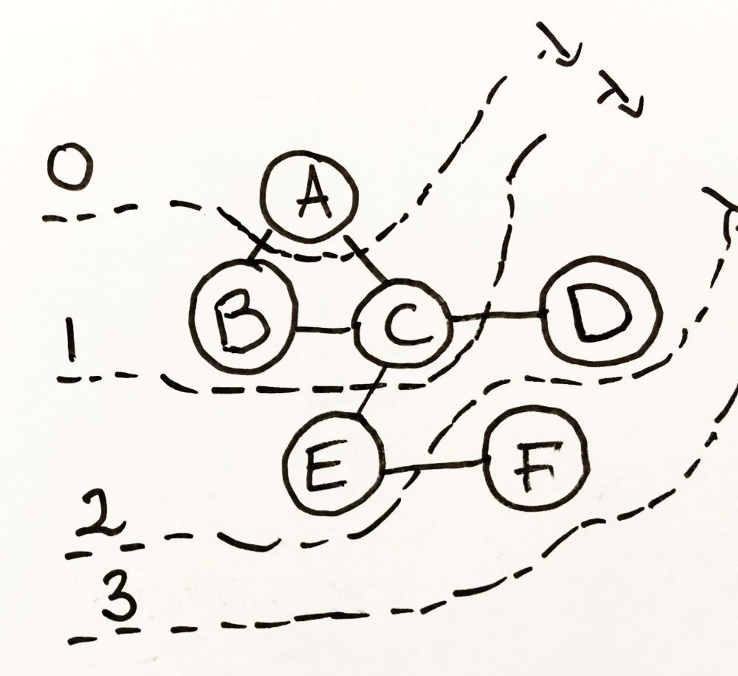
Breadth First Search



Starting Point

Pseudo code

```
visited = []; currentLevel = [root]
for currentLevel has nodes {
    nextLevel = []

    for each node in currentLevel {
        for neighbor in node.neighbors {
            if visited.contains neighbor {
                continue
            }

            nextLevel.add neighbor
            visited.add   neighbor
        }
    }
    currentLevel = nextLevel
```

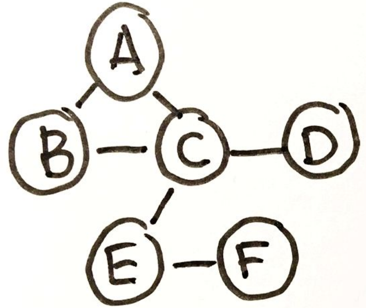


```go
func BreadthFirst(g *graph.Graph, source graph.Node, level []int)
{
    visited := NewNodeSet(g.NodeCount())

    currentLevel := make([]graph.Node, 0, g.NodeCount())
    nextLevel := make([]graph.Node, 0, g.NodeCount())

    level[source] = 1
    visited.Add(source)
    currentLevel = append(currentLevel, source)

    levelNumber := 2

    for len(currentLevel) > 0 {
        for _, node := range currentLevel {
            for _, neighbor := range g.Neighbors(node) {
                if !visited.Contains(neighbor) {
                    nextLevel = append(nextLevel, neighbor)
```

```go
    for len(currentLevel) > 0 {
        for _, node := range currentLevel {
            for _, neighbor := range g.Neighbors(node) {
                if !visited.Contains(neighbor) {
                    nextLevel = append(nextLevel, neighbor)
                    level[neighbor] = levelNumber
                    visited.Add(neighbor)
                }
            }
        }

        levelNumber++
        currentLevel = currentLevel[:0:cap(currentLevel)]
        currentLevel, nextLevel = nextLevel, currentLevel
    }
}
```

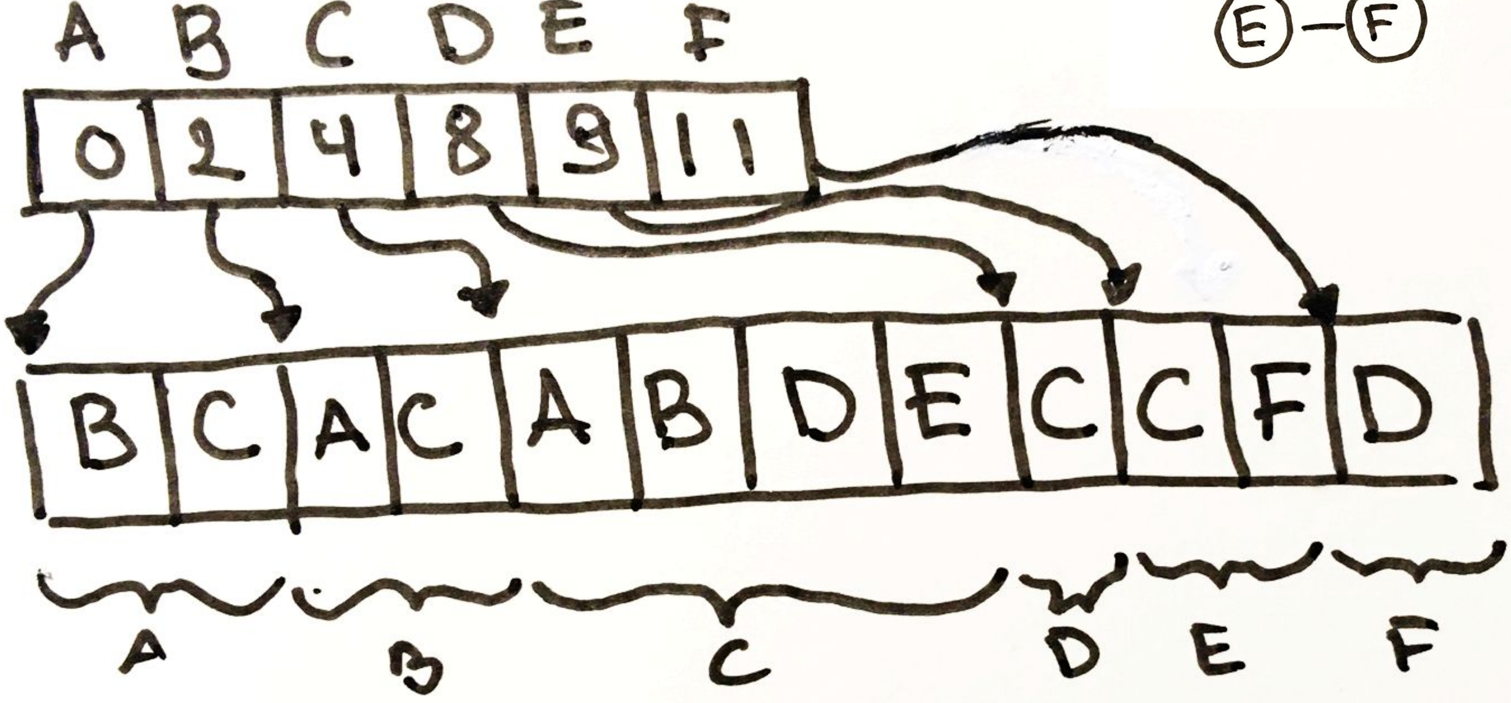
```go
const bucket_bits = 5
const bucket_size = 1 << 5
const bucket_mask = bucket_size - 1

type NodeSet []uint32

func (set NodeSet) Offset(node graph.Node) (bucket, bit uint32) {
    bucket = uint32(node >> bucket_bits)
    bit = uint32(1 << (node & bucket_mask))
    return bucket, bit
}

func (set NodeSet) Add(node graph.Node) {
    bucket, bit := set.Offset(node)
    set[bucket] |= bit
}

func (set NodeSet) Contains(node graph.Node) bool {
```

Graph: Compact Adjacency List



Span

List



Graph: Compact Adjacency List

```
type Node = uint32

type Graph struct {
    List []Node
    Span []uint64
}

func (graph *Graph) Neighbors(n Node) []Node {
    start, end := graph.Span[n], graph.Span[n+1]
    return graph.List[start:end]
}
```

Why not pointers?

```
type Node struct {
    Neighbors []*Node
}
```

Back of the envelope calculation...

```
type Node struct {
    Neighbors []*Node
}
```

```
number_of_nodes * 24 bytes +
number_of_edges * 8 bytes
```

```
type Node = uint32

type Graph struct {
    List []Node
    Span []uint64
}
```

```
number_of_edges * 4 bytes +
number_of_nodes * 8 bytes +
2 * 24
```

```
type Node struct {
    Neighbors []*Node
}
```

```
type Node uint32

type Graph struct {
    List []Node
    Span []uint64
}
```

```
number_of_nodes * 24 bytes +
number_of_edges * 8 bytes
```

```
number_of_edges * 4 bytes +
number_of_nodes * 8 bytes +
2 * 24
```

15.96GB (but actually larger)

7.72GB

for
65 million nodes and 1.8 billion edges

To optimizing!!!


i7-2820QM | 5M nodes
65M nodes | E5-2670v3
10s 9s 8s 7s 6s 5s 4s 3s 2s 1s 0s
0s 5s 10s 15s 20s 25s 30s 35s 40s 45s 50s
2.29s
baseline
48.44s

Easy things first: **-gcflags all=-B**
disable bounds checks

```
...
if !visited.Contains(neighbor) {
    nextLevel = append(nextLevel, neighbor)
    level[neighbor] = levelNumber
    visited.Add(neighbor)
}
...

func (graph *Graph) Neighbors(n Node) []Node {
    start, end := graph.Span[n], graph.Span[n+1]
    return graph.List[start:end]
}
```

Easy things first: **-gcflags all=-B**
disable bounds checks

no significant difference

Easy things first: **go.tip**

Maybe by waiting for next release
you don’t have to optimize?

10% faster

Easy things first: **Tweak GC**

GOGC=50

GOGC=100

GOGC=150

GOGC=200

GOGC=off

didn’t try

Reusing Level

```
for len(currentLevel) > 0 {
    for _, node := range currentLevel {
        for _, neighbor := range g.Neighbors(node) {
            if !visited.Contains(neighbor) {
                nextLevel = append(nextLevel, neighbor)
                level[neighbor] = levelNumber
                visited.Add(neighbor)
            }
        }
    }
```

Reusing Level

```
for len(currentLevel) > 0 {
    for _, node := range currentLevel {
        for _, neighbor := range g.Neighbors(node) {
            if level[neighbor] == 0 {
                nextLevel = append(nextLevel, neighbor)
                level[neighbor] = levelNumber
            }
        }
    }
```

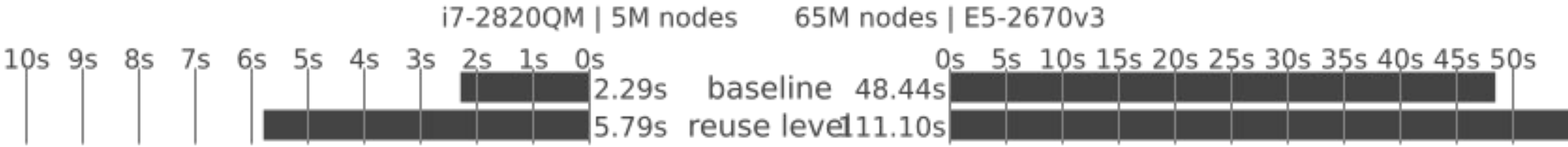
Reusing Level

10% faster...

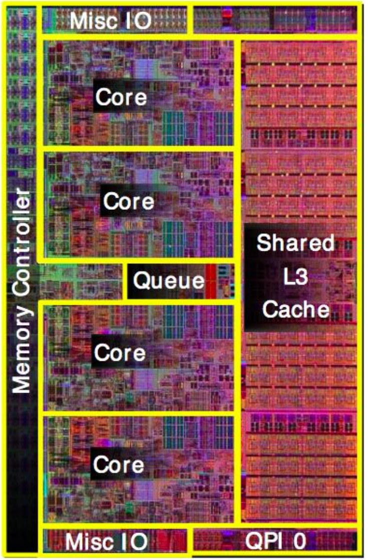
Dataset	sg-10k-250k			
Processor	i7–2820QM		i5–5257U	
Flags		-B		-B
baseline	2.65	2.66	2.29	2.27
reuse level	2.20	2.10	2.06	1.94

Reusing Level

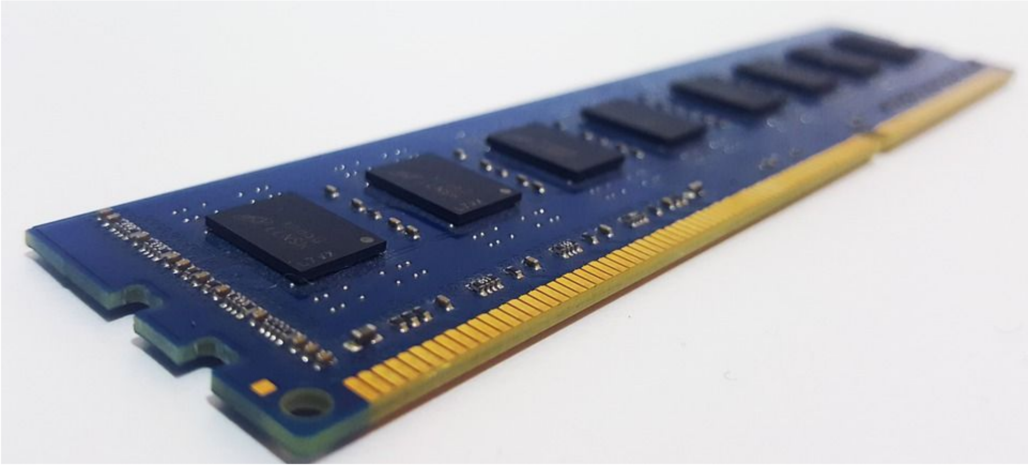
10% faster... and...
2x slower for large-dataset



Reusing Level



https://commons.wikimedia.org/wiki/File:Shared_cache_cpu.png

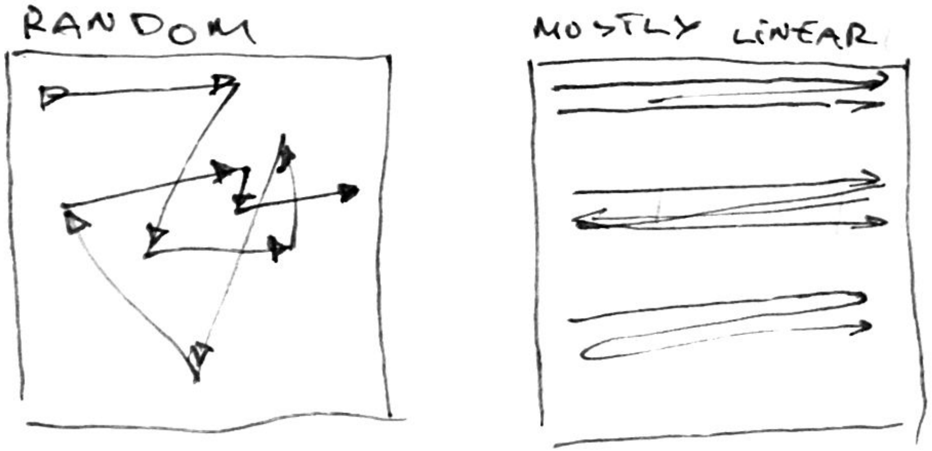


https://pixabay.com/photos/memory-ram-random-access-memory-4704236/

Measure - VTune

`for len(currentLevel) > 0 {`		
`    for _, node := range currentLevel {`	0.000s	0.066s
`        for _, neighbor := range g.Neighbors(node) {`	0.016s	8.882s
`            if !visited.Contains(neighbor) {`	0.000s	0.320s
`                nextLevel = append(nextLevel, neighbor)`	0.000s	0.141s
`                level[neighbor] = levelNumber`	0.000s	0.006s
`                visited.Add(neighbor)`		
`            }`		
`        }`		
`    }`		

Random Access



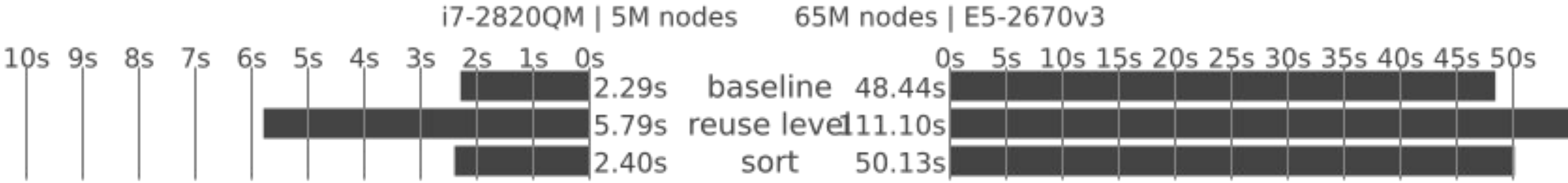
`for len(currentLevel) > 0 {`		
`    for _, node := range currentLevel {`	0.000s	0.066s
`        for _, neighbor := range g.Neighbors(node) {`	0.016s	8.882s
`            if !visited.Contains(neighbor) {`	0.000s	0.320s
`                nextLevel = append(nextLevel, neighbor)`	0.000s	0.141s
`                level[neighbor] = levelNumber`	0.000s	0.006s
`                visited.Add(neighbor)`		

Random Access

```
            nextLevel = append(nextLevel, neighbor)
            level[neighbor] = levelNumber
            visited.Add(neighbor)
        }
    }
}

sort.Slice(nextLevel, func(i, k int) bool {
    return nextLevel[i] < nextLevel[k]
})
```

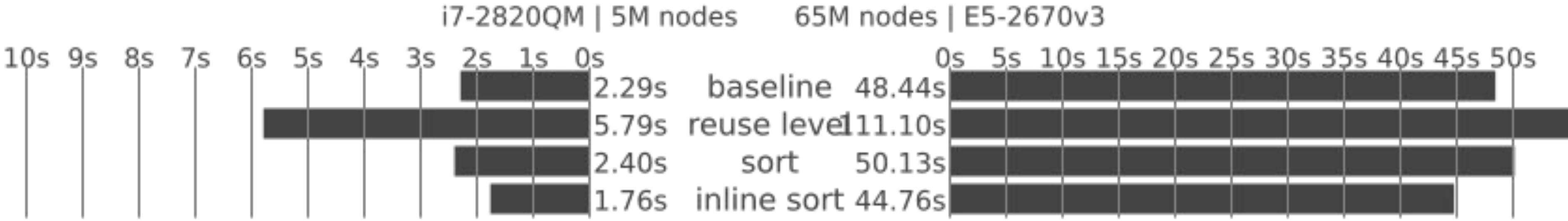
```go
sort.Slice(nextLevel, func(i, k int) bool {
    return nextLevel[i] < nextLevel[k]
})
```

~1% difference



`optimized sort uint32`

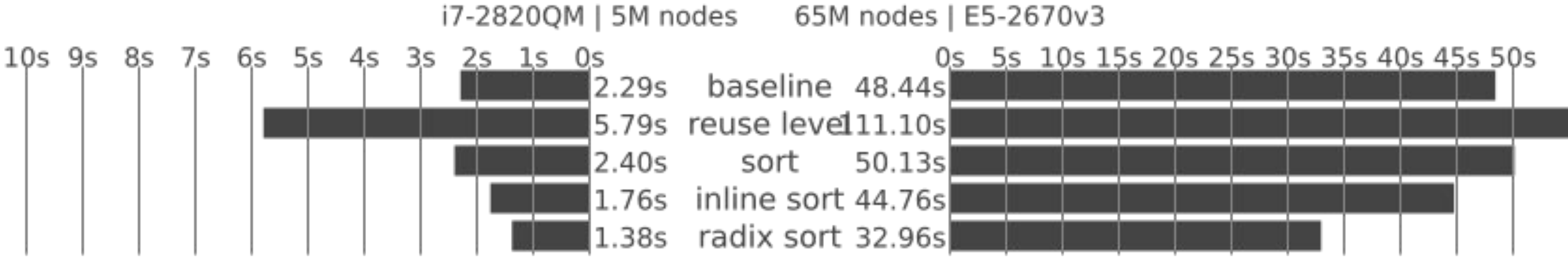
~20% improvement



radix sort

suggested by Damian Gryski at #performance

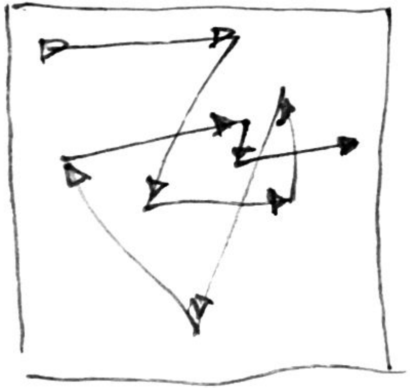
+5% improvement



Random Access Part 2

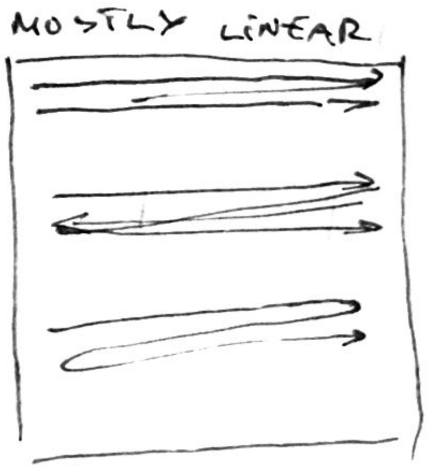
```
for _, node := range currentLevel {
    for _, neighbor := range g.Neighbors(node) {
        if !visited.Contains(neighbor) {
            nextLevel = append(nextLevel, neighbor)
            level[neighbor] = levelNumber
            visited.Add(neighbor)
        }
    }
}

zuint32.Sort(nextLevel)
```

RANDOM



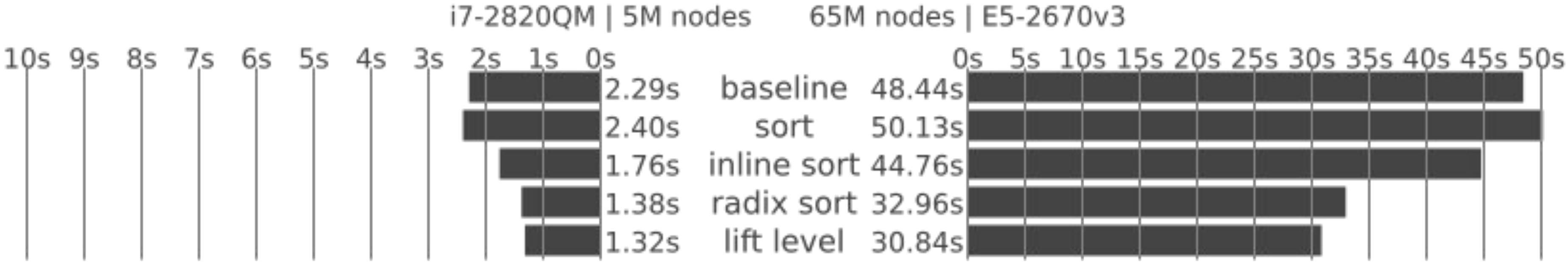
Random Access Part 2

```
for _, node := range currentLevel {
    for _, neighbor := range g.Neighbors(node) {
        if !visited.Contains(neighbor) {
            nextLevel = append(nextLevel, neighbor)
            visited.Add(neighbor)
        }
    }
}

zuint32.Sort(nextLevel)
for _, neighbor := range nextLevel {
    level[neighbor] = levelNumber
}
```



Random Access Part 2

~15% improvement



Slow visiting set

`for len(currentLevel) > 0 {`		
`    for _, node := range currentLevel {`	0.000s	0.040s
`        for _, neighbor := range g.Neighbors(node) {`	0.001s	2.991s
`            if !visited.Contains(neighbor) {`	0.000s	0.389s
`                nextLevel = append(nextLevel, neighbor)`	0.000s	0.157s
`                visited.Add(neighbor)`		
`            }`		
`        }`		
`    }`		

Maybe order of operations?

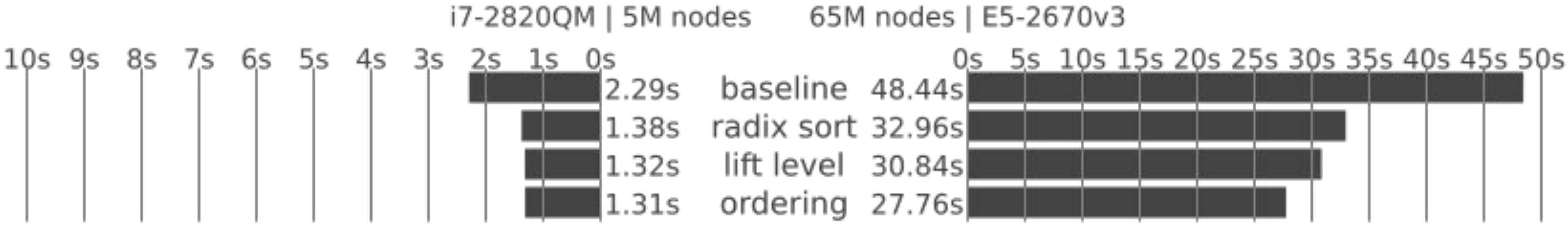
```
if !visited.Contains(neighbor) {
    nextLevel = append(nextLevel, neighbor)
    visited.Add(neighbor)
}
```

vs.

```
if !visited.Contains(neighbor) {
    visited.Add(neighbor)
    nextLevel = append(nextLevel, neighbor)
}
```

Maybe order of operations?

~10% improvement from order

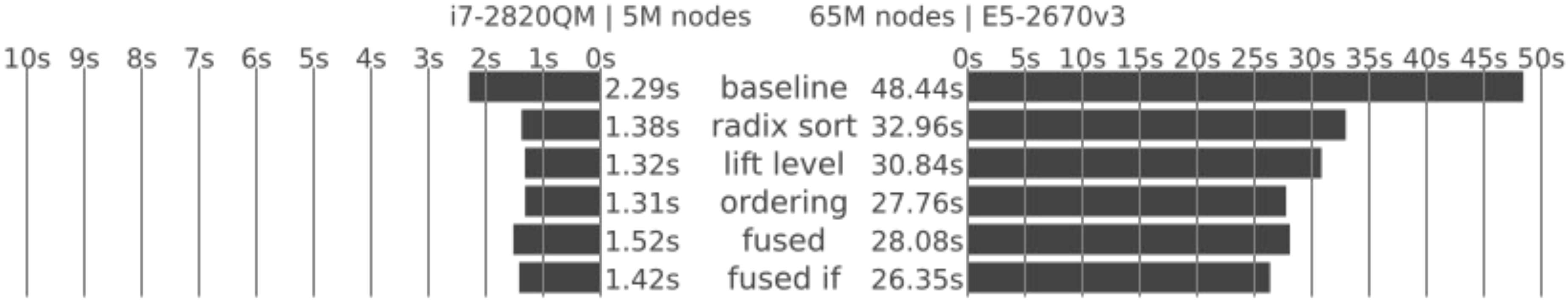


fuse Contains + Add = TryAdd?

```
func (set NodeSet) TryAdd(node graph.Node) bool {
    bucket, bit := set.Offset(node)
    empty := set[bucket]&bit == 0
    set[bucket] |= bit
    return empty
}

func (set NodeSet) TryAdd(node graph.Node) bool {
    bucket, bit := set.Offset(node)
    empty := set[bucket]&bit == 0
    if empty {
        set[bucket] |= bit
    }
    return empty
}
```

Fusing Contains + Add = TryAdd

slight improvement



Maybe deduplicate later?

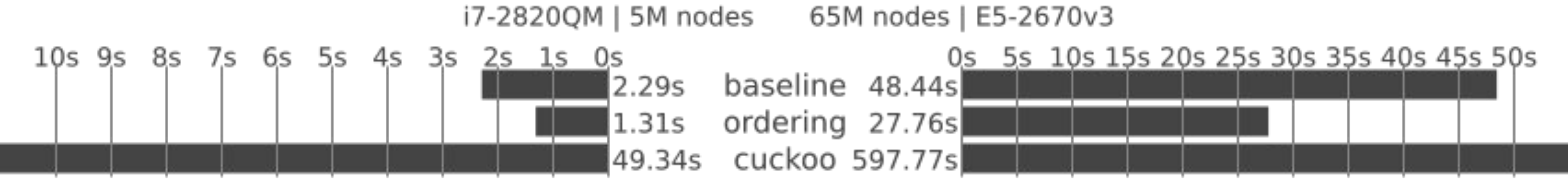
```
for _, node := range currentLevel {
    for _, neighbor := range g.Neighbors(node) {
        nextLevel = append(nextLevel, neighbor)
    }
}
zuint32.Sort(nextLevel)
for _, neighbor := range nextLevel {
    if !visited.Contains(neighbor) {
        level[neighbor] = levelNumber
        visited.Add(source)
    } else {
        ...
```

terrible idea (next level was 10x larger)

Cuckoo and Bloom Filter?

Cuckoo and Bloom Filter?

terrible idea #2
20x slower without implementing everything

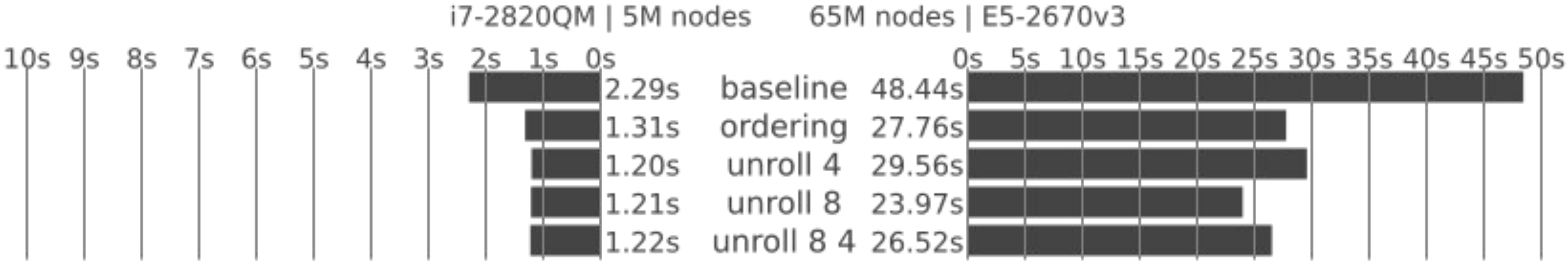


Unrolling loops

```
neighbors := g.Neighbors(node)

for ; i < len(neighbors)-1; i += 2 {
    n1, n2, n3, n4 := neighbors[i], neighbors[i+1], ...

    if !visited.Contains(n1) {
        visited.Add(n1)
        nextLevel = append(nextLevel, n1)
    }

    if !visited.Contains(n2) {
        visited.Add(n2)
        nextLevel = append(nextLevel, n2)
    }
```

Unrolling loops

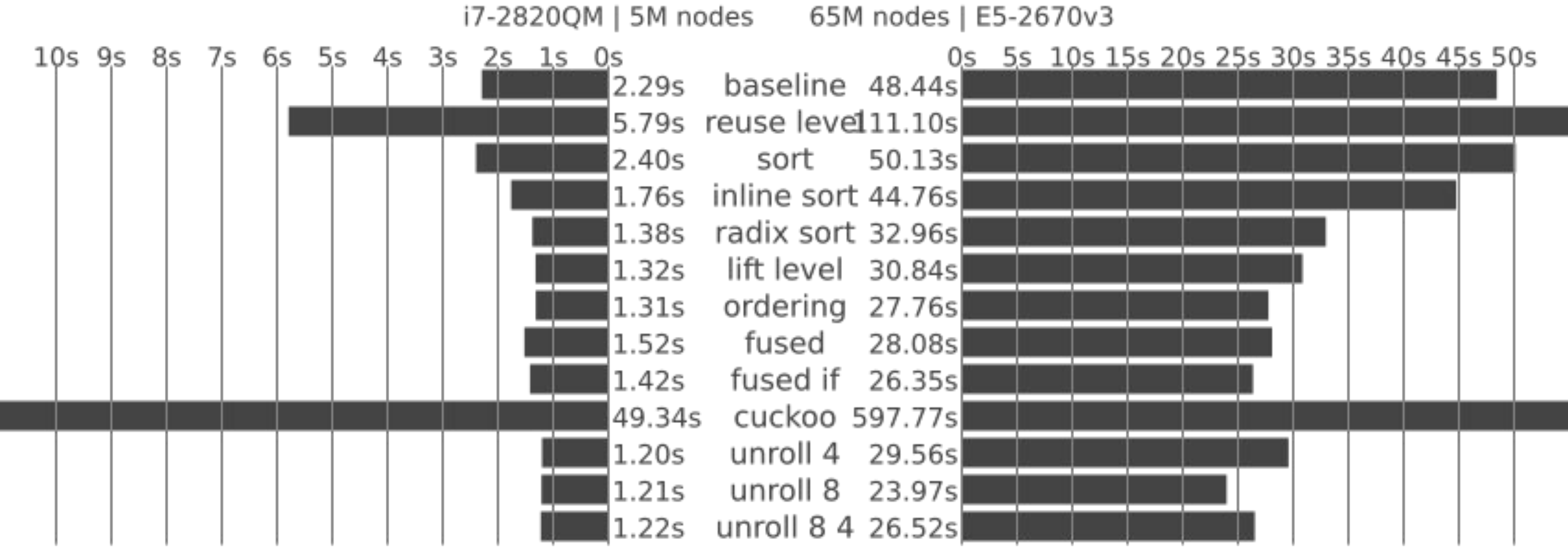
~5% slight improvement



Summary so far...

Summary so far... **48s -> 24s**

Better than the C++ version.



Going Parallel

Knowledge helps

<u>The Little Book of Semaphores</u>

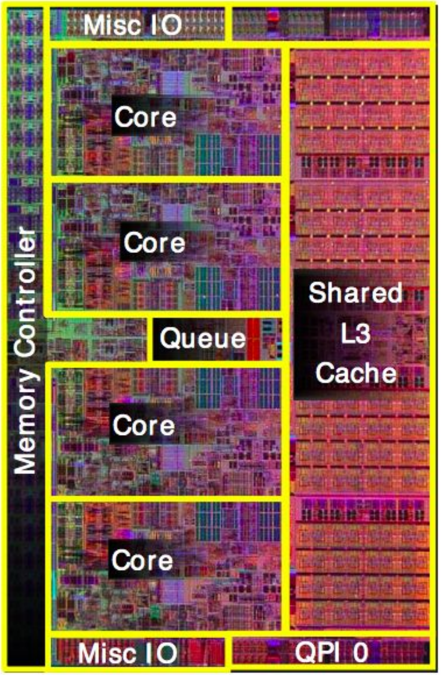
by Allen B. Downey

Basic ideas for modern processor parallelism

Distribution of work takes time.

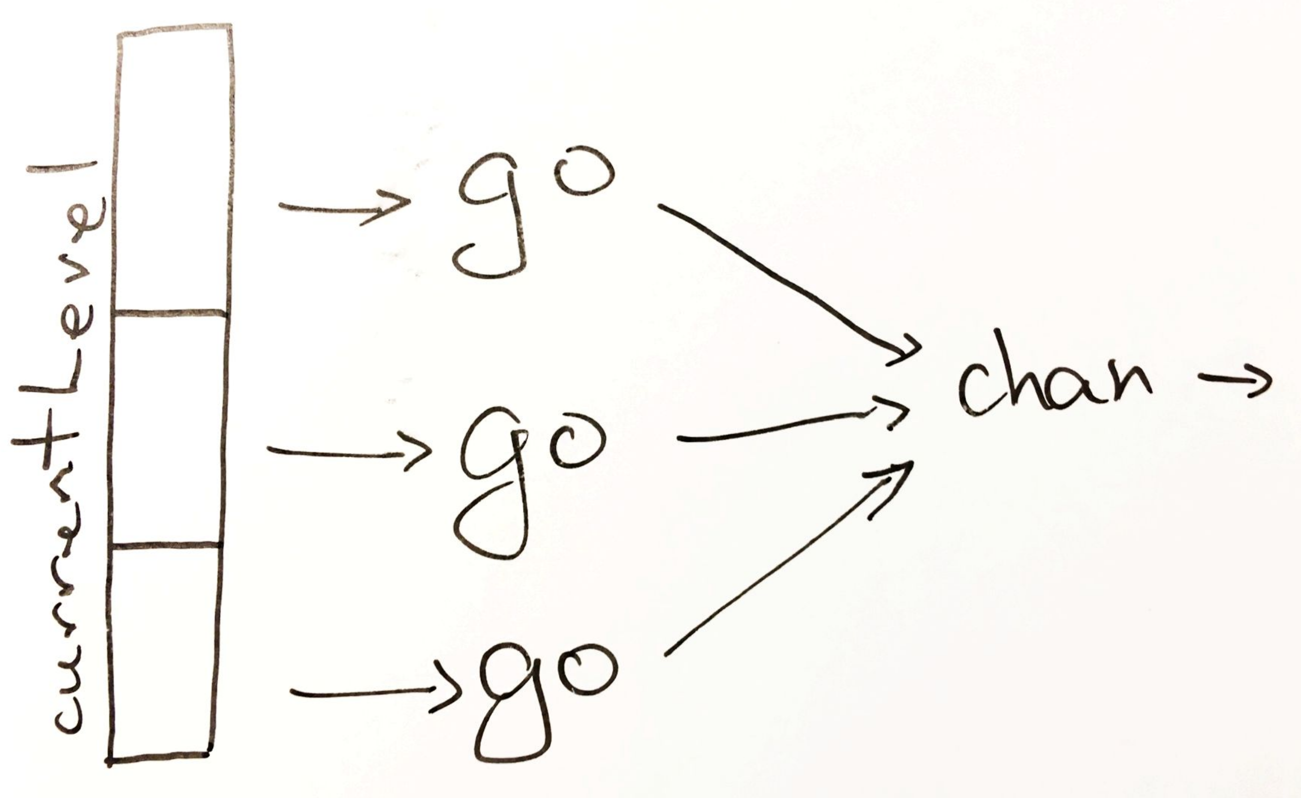
Collection of work takes time.

Don’t change data in the same memory location.



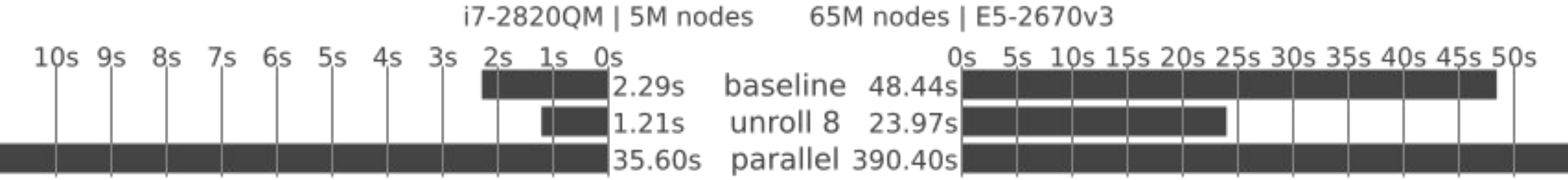
https://commons.wikimedia.org/wiki/File:Shared_cache_cpu.png

First Try

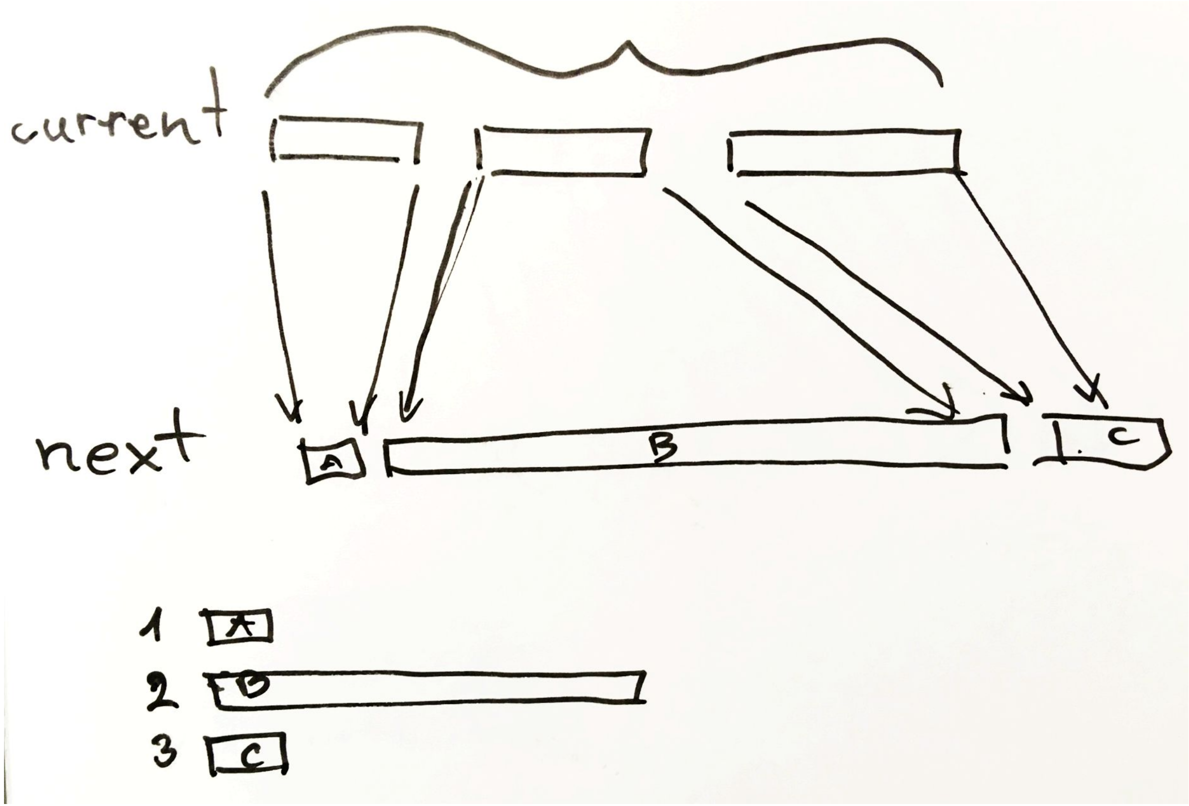


First Try

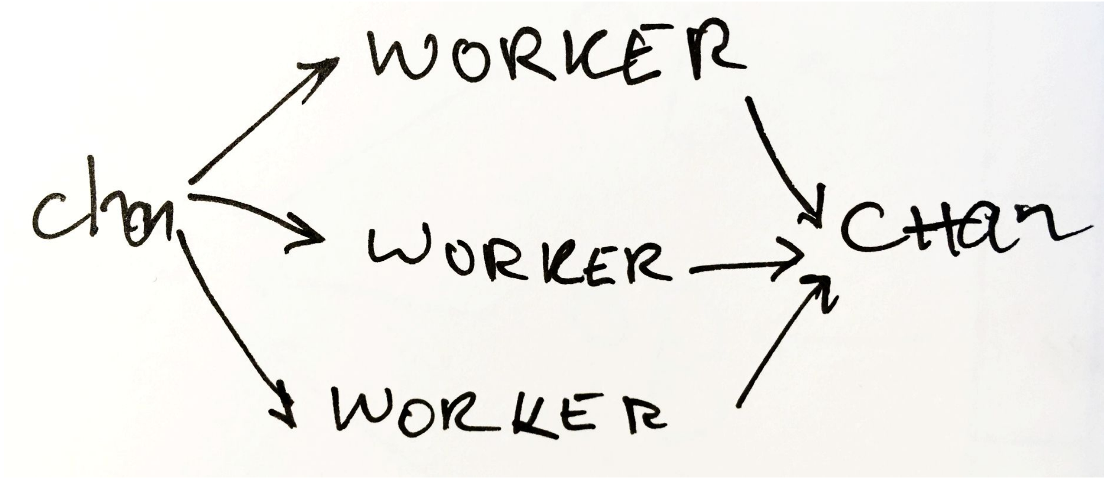
over 10x slower



First Try? But why?



Channels

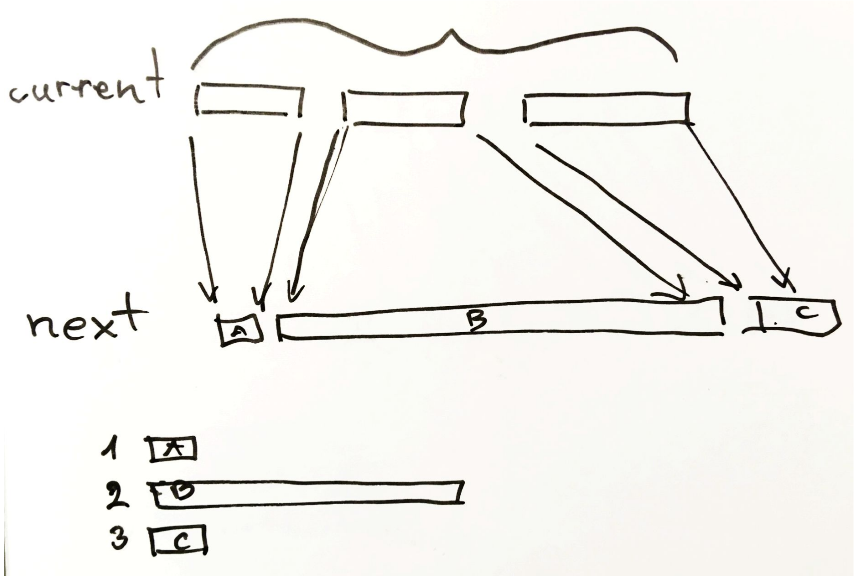


Channels

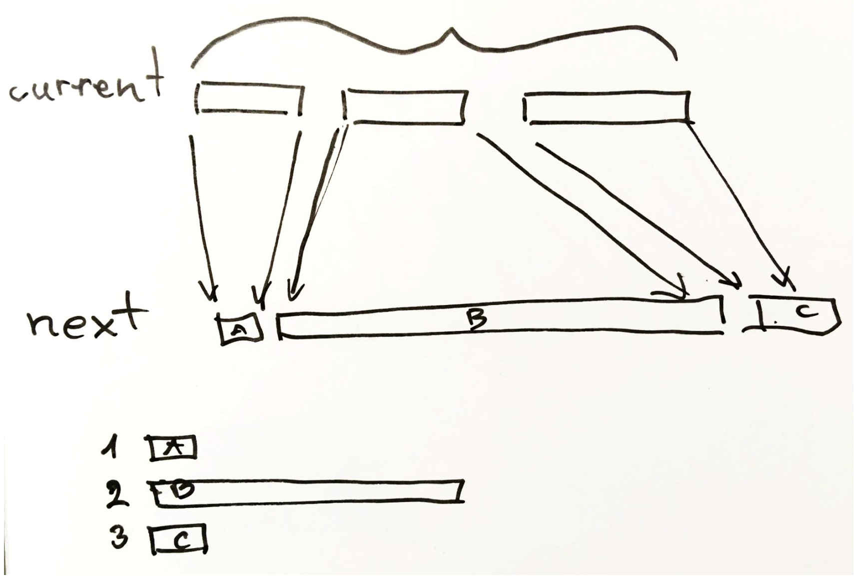
faster, but just barely

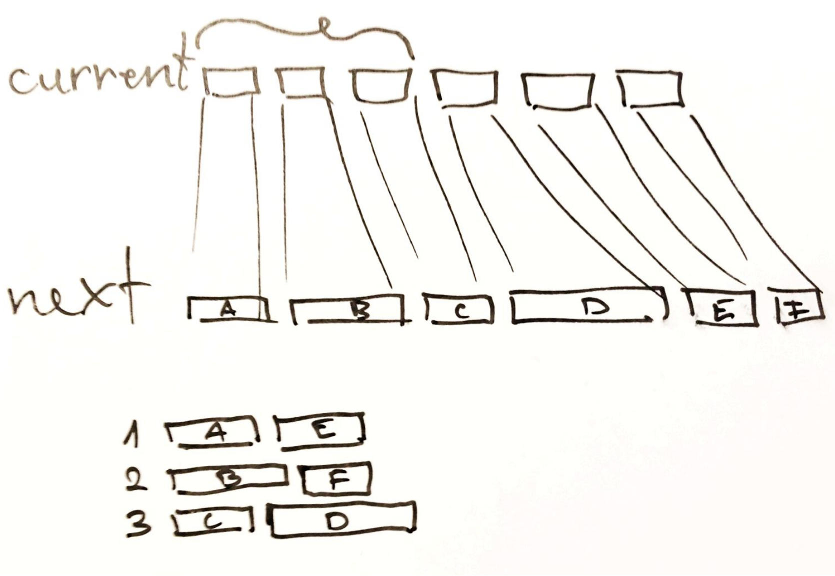
sg-5m-100m	baseline	2326.9
sg-5m-100m	ordering	1161.37
sg-5m-100m	parallel	21741.95
sg-5m-100m	parchan 4x	1564.43
sg-5m-100m	parchan 8x	1998.48

Fairer work distribution



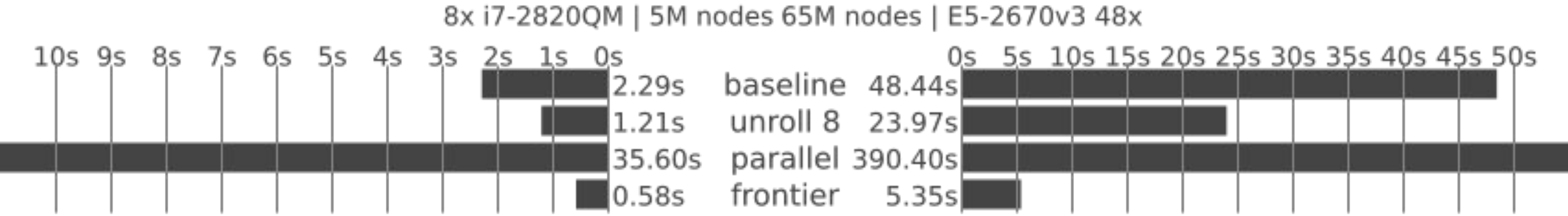
Fairer work distribution





Fairer work distribution + without channels

2x faster on 4 cores
4x faster on 48 cores



Bottlenecks

```
for len(currentLevel.Nodes) > 0 {
    async.Run(procs, func(i int) {
        process(g, currentLevel, nextLevel, visited)
    })

    zuint32.Sort(nextLevel)
...
}
```

**Empty programs give
the wrong answer in no time at all.**

**It's easy to be fast
if you don't have to be correct.**

We can sort without caring about the result

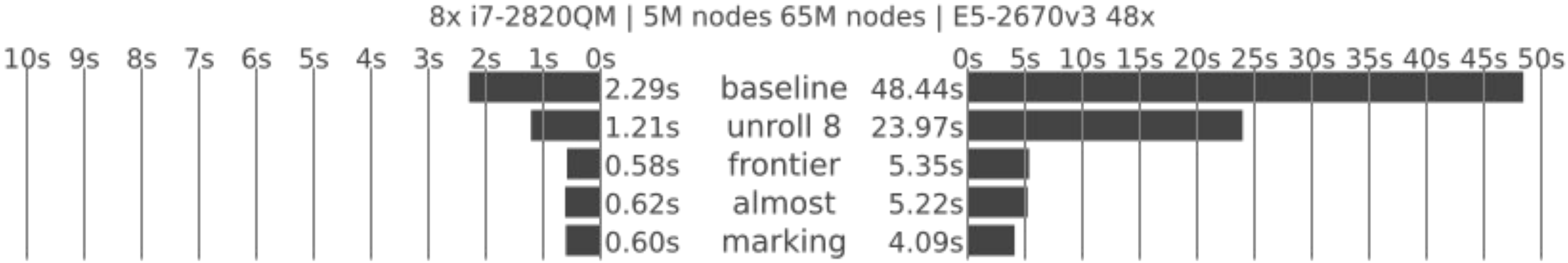
```
for len(currentLevel.Nodes) > 0 {
    async.Run(procs, func(i int) {
        process(g, currentLevel, nextLevel, visited)
    })

    async.BlockIter(nextLevel.count, procs,
        func(low, high int) {
            zuint32.Sort(nextLevel.Nodes[low:high])
        })

    for _, neighbor := range nextLevel.Nodes {
        level[neighbor] = levelNumber
    }
```

We can sort without caring about the result

```
for len(currentLevel.Nodes) > 0 {
    async.Run(procs, func(i int) {
        process(g, currentLevel, nextLevel, visited)
    })

    async.BlockIter(nextLevel.count, procs,
        func(low, high int) {
            zuint32.Sort(nextLevel.Nodes[low:high])

            for _, neighbor := range nextLevel.Nodes[low:high] {
                level[neighbor] = levelNumber
            }
        })
```

Almost sorting in parallel

~10% faster



Visited set is still slow

```
type NodeSet []uint32

func (set NodeSet) TryAdd(node graph.Node) bool {
    bucket, bit := set.Offset(node)
    empty := set[bucket]&bit == 0
    if empty {
        set[bucket] |= bit
    }
    return empty
}
```

Prefetch buckets in bulk

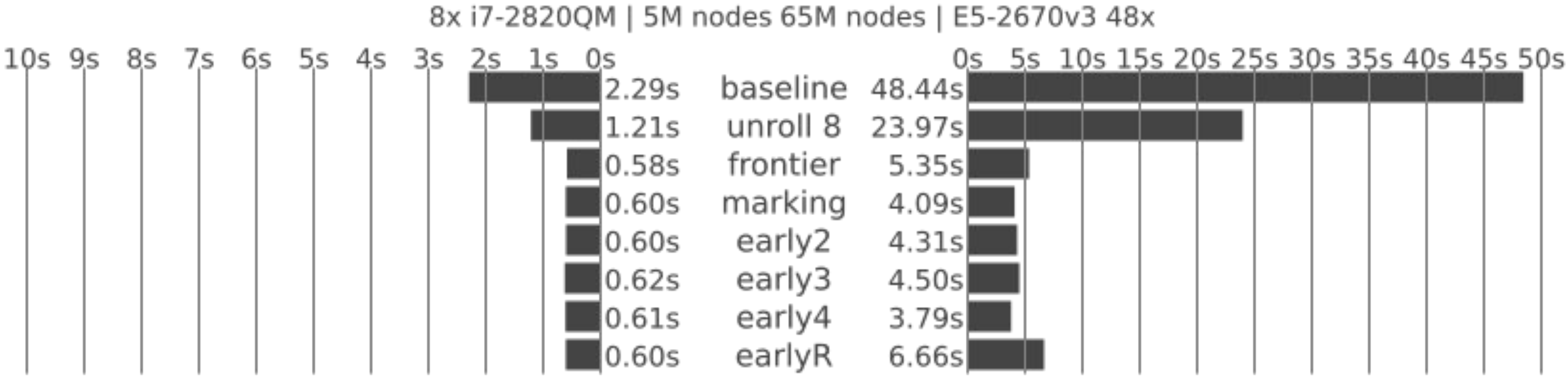
```
type NodeSet []uint32

func (set NodeSet) GetBuckets4(a, b, c, d graph.Node)
    (x, y, z, w uint32) {
    x = set[a>>bucket_bits]
    y = set[b>>bucket_bits]
    z = set[c>>bucket_bits]
    w = set[d>>bucket_bits]
    return
}
```

Prefetch buckets in bulk

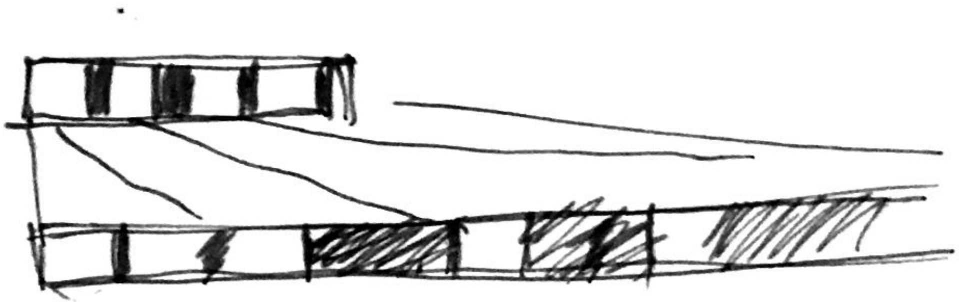
```
for ; i < len(neighbors)-3; i += 4 {
    n1, n2, n3, n4 := neighbors[i], neighbors[i+1], ...

    x1, x2, x3, x4 := visited.GetBuckets4(n1, n2, n3, n4)

    if visited.TryAddFrom(x1, n1) {
        nextLevel.Write(&writeLow, &writeHigh, n1)
    }
    if visited.TryAddFrom(x2, n2) {
        nextLevel.Write(&writeLow, &writeHigh, n2)
    }
```

Fetch buckets in bulk

~10% improvement



Multilevel bitmaps



Multilevel bitmaps

aka. Let's add a cache to your cache.

10% slower

Long live workers

Maybe let’s not create a new `goroutine` every loop?

Long live workers

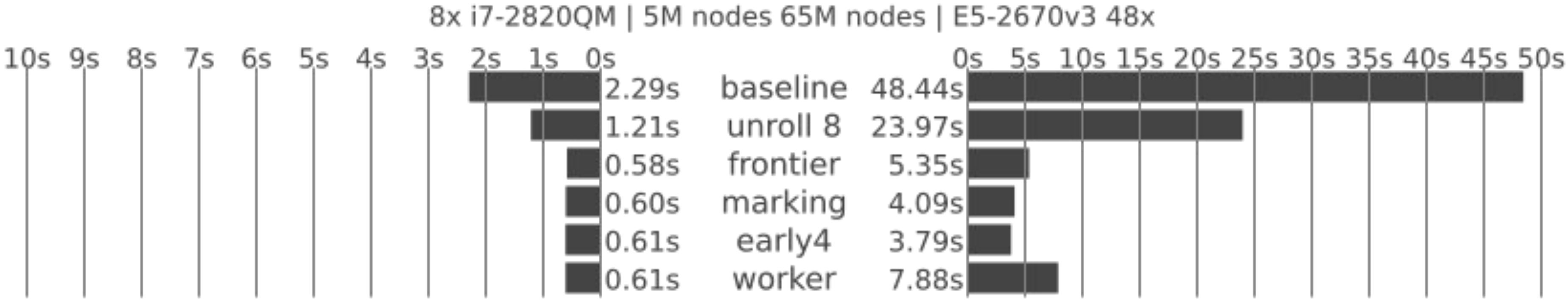
Maybe don’t spin up a new `goroutine` every loop?

Implementation was a pain... also, made mistakes:

sync: WaitGroup misuse: Add called concurrently with Wait

Long live workers

20% slower on 4 cores
2x slower on 48 cores



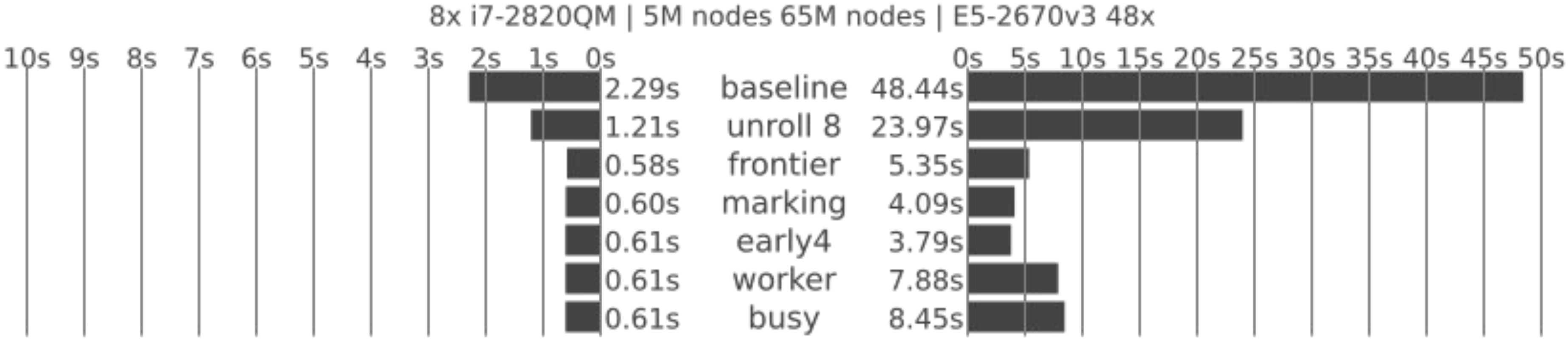
Busy Group

[DO NOT USE IN PROD]

```
type BusyGroup struct{ sema int32 }

func (bg *BusyGroup) Add(v int) {
    atomic.AddInt32(&bg.sema, int32(v))
}
func (bg *BusyGroup) Done() {
    bg.Add(-1)
}
func (bg *BusyGroup) Wait() {
    for atomic.LoadInt32(&bg.sema) != 0 {
        runtime.Gosched()
    }
}
```

Busy Group

10% slower on 4 cores
2x slower on 48 cores

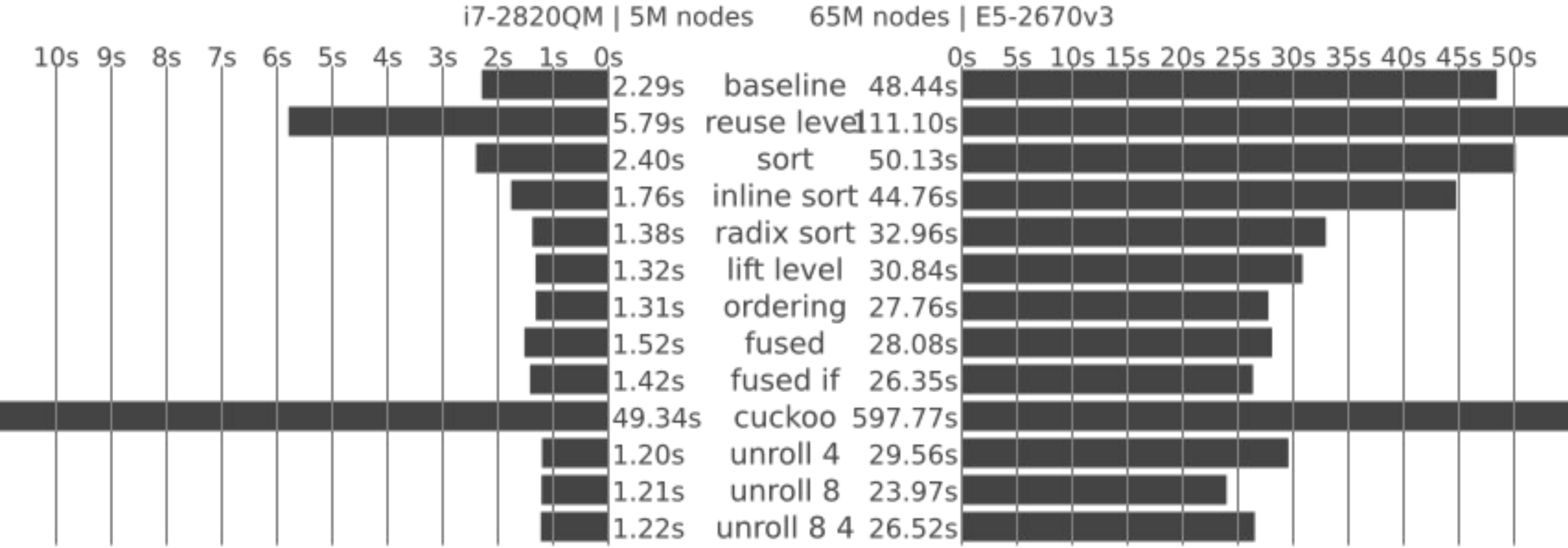


and this is how optimization usually ends

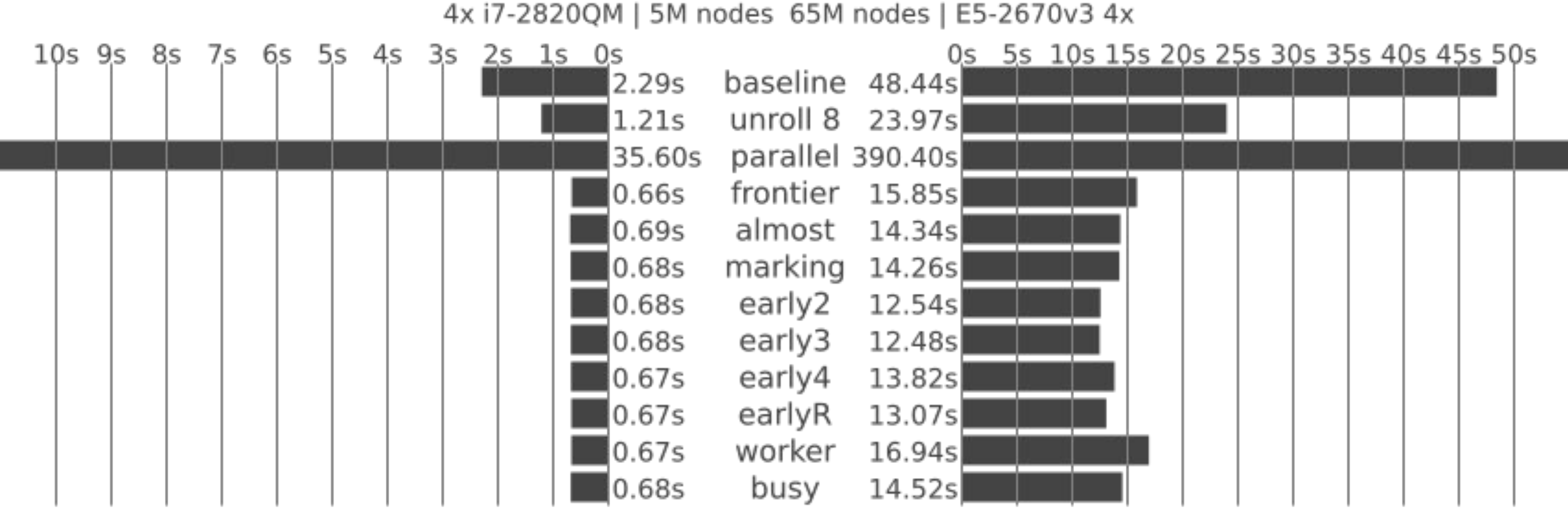
with a bunch of failures

Final Results

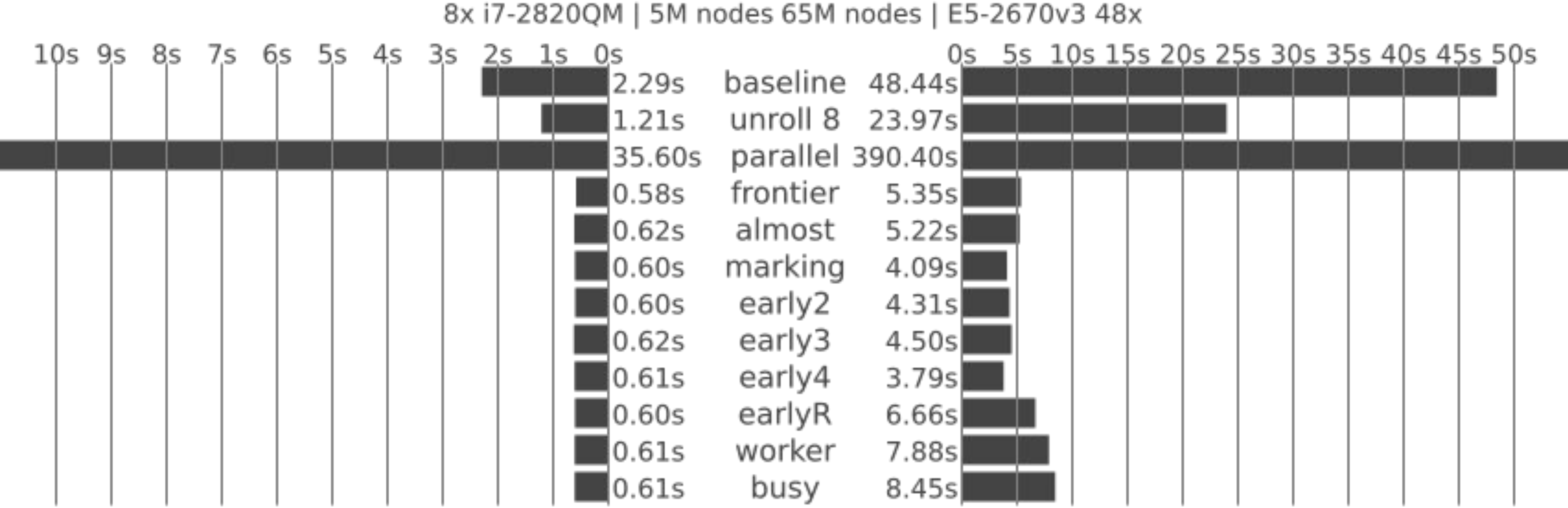
Single Core: 48s -> **24s**



4x Core: 48s -> 24s -> **12.5s**



48x Core: 48s -> 24s -> **3.8s**



There’s at least 20% improvement somewhere…

But, that’s for another day.

Thanks for listening…

Thanks for listening...

https://egonelbre.com/a-tale-of-bfs/

https://egonelbre.com/a-tale-of-bfs-going-parallel/