Node.js와 Go의 멀티쓰레딩 환경

한우석 / Golang Korea

Speaker

한우석

아카데미소프트 / GDG Golang Korea

Multipotentialite

Stay hungry, stay foolish.

Index

이번 발표에서는

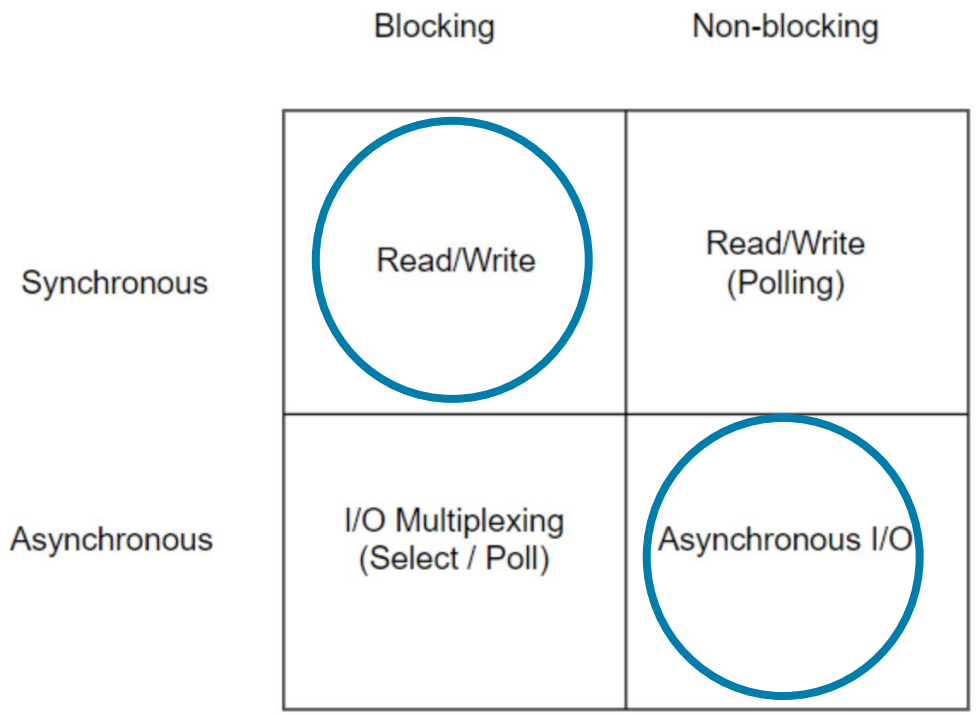
다루지 않을 거에요

- Node.js와 Go의 멀티쓰레딩 환경 구축에 대한 Deep Dive

다룰거에요

- Node.js와 Go의 멀티쓰레딩 환경에 대한 전반적인 이해
- Cpu intensive, I/O Intensive 한 상황에서의 Node.js, Go의 벤치마킹 결과

Benchmark Situation

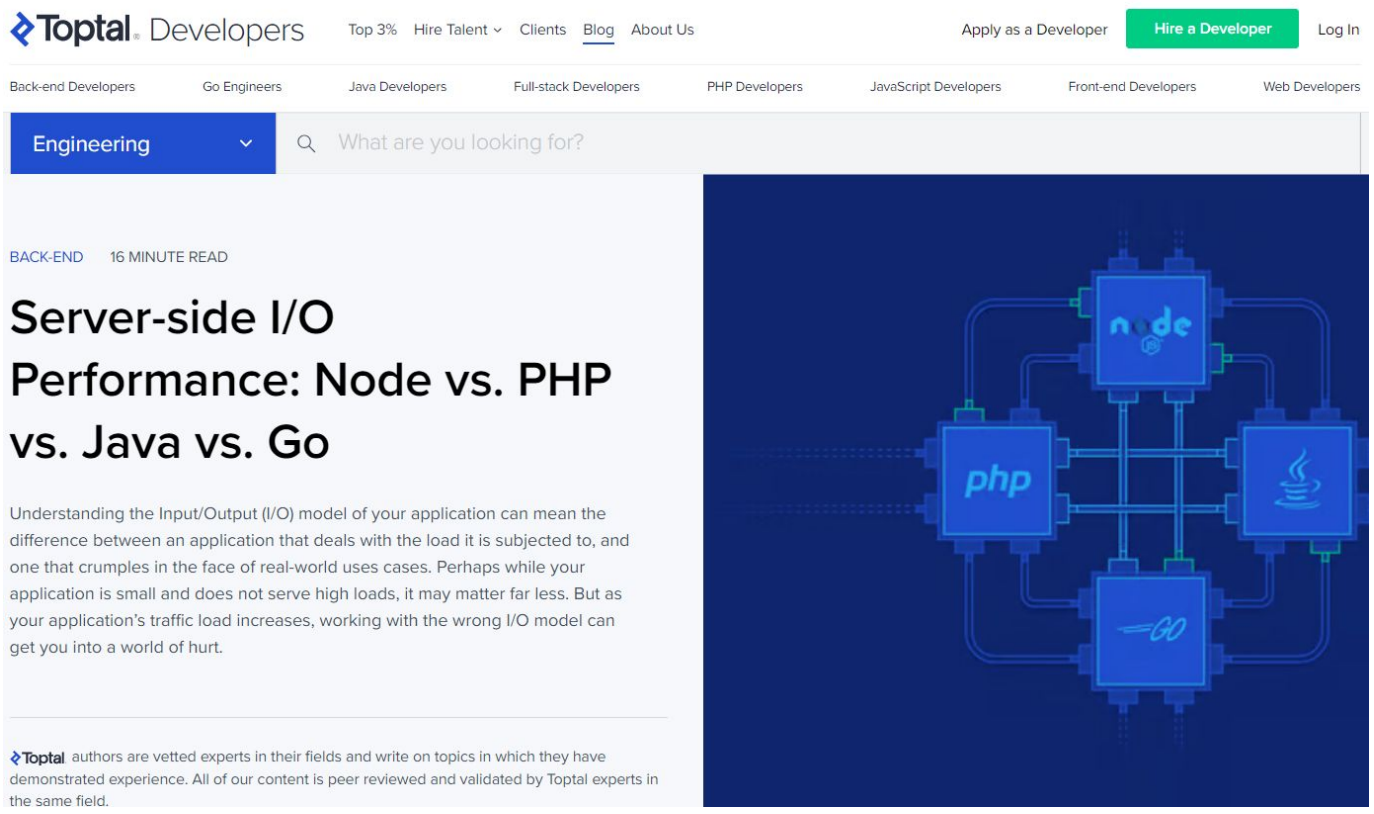


Why Node.js & Go?

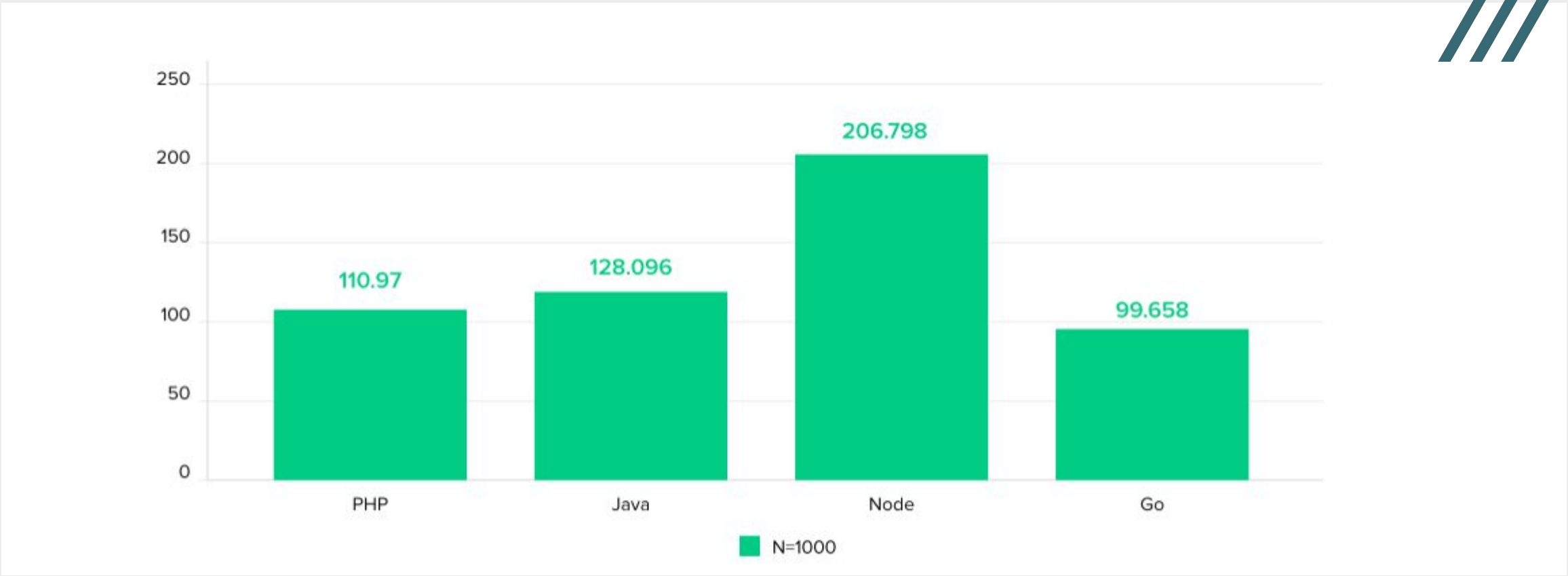
미안하다...

오호라…

https://www.toptal.com/back-end/server-side-io-performance-node-php-java-go



cpu bound한 요청을 처리하는데 1000개당 걸린 시간 (적을수록 좋음!)



그런데…

저거 실험 결과가….좀.

Node.js의 버전이 낮은건 둘째치고 (다른것들도 낮은것 같음.)

같은 머신이라고 했을 때

배포 시 cpu 놀지 말라고 클러스터링 셋팅하고 하는데

그에 대한 말이 없네요.

다른 언어들은 전부 멀티스레드 사용하는데

노드만 서버에서 자원하나만 돌린것 같은데요.

맞다면 너무 편파적인것 같습니다.

cpu집중 작업은 당연히 성능 딸리겠지만…

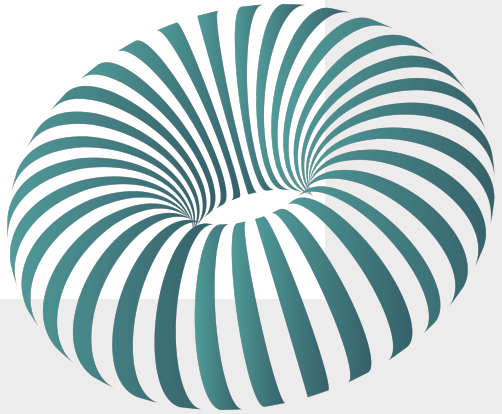
노드는 pm2등으로 클러스터링이된 결과인가요?

Did not found any word on how Node.js is running in your benchmarks.
I mean, did you use clustering(e.g. run `pm2 start index.js -i 0` to use all CPUs) ?
If not, then we could consider this benchmark as unfair for NodeJS, because Go uses all CPUs for his routines

Use a nodejs cluster at least! How can you compare a multicore program with a single thread execution? Everybody uses nodejs clusters in production! This benchmark means nothing to me!

하지만…

Node.js와 Thread



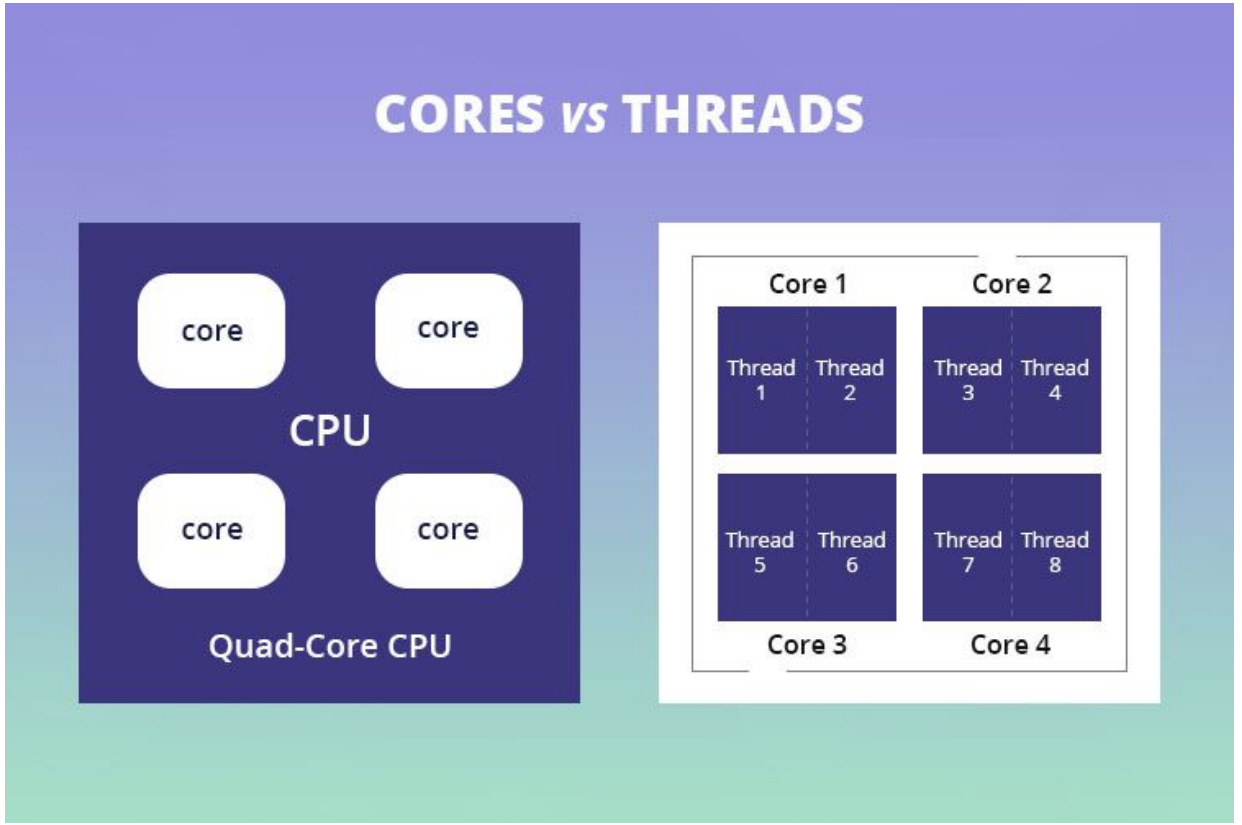
What is Node.js?

Node.js는 크로스플랫폼 오픈소스 자바스크립트 런타임 환경으로 윈도우, 리눅스, macOS 등을 지원한다. Node.js는 V8 자바스크립트 엔진으로 구동되며, 웹 브라우저 바깥에서 자바스크립트 코드를 실행할 수 있다.

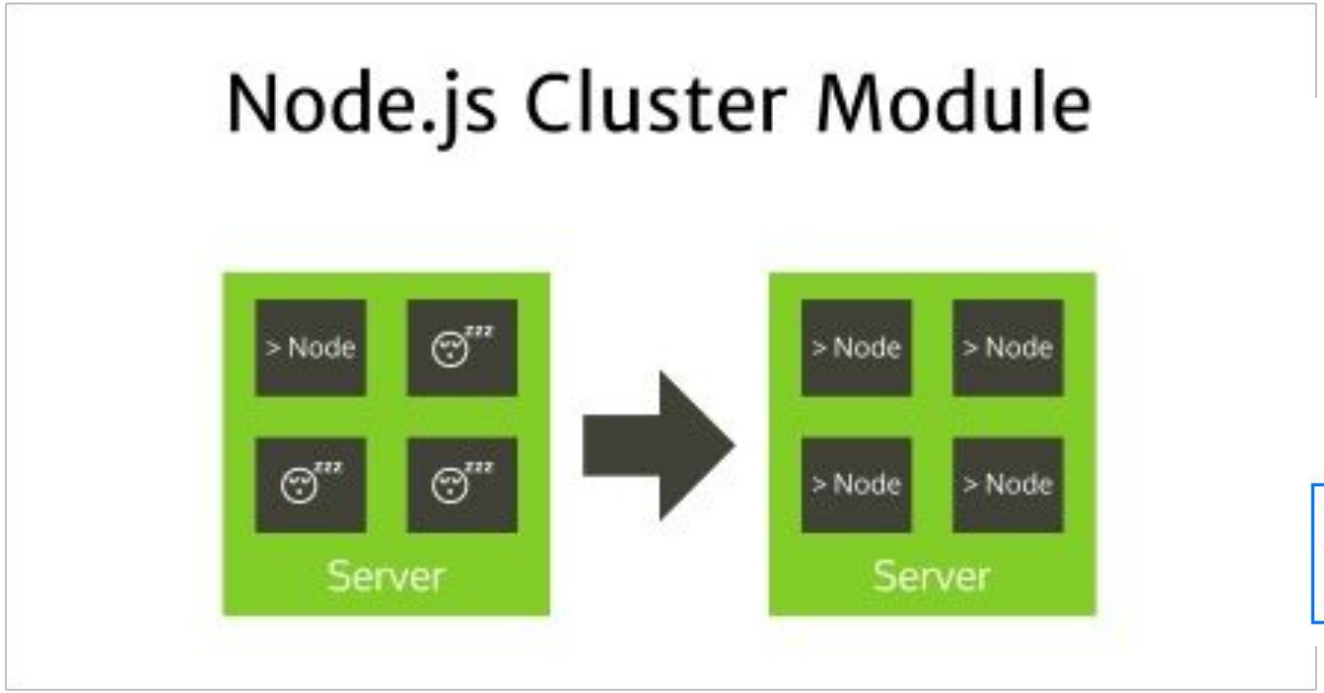
주로 확장성 있는 네트워크 애플리케이션과 서버 사이드 개발에 사용되는 소프트웨어 플랫폼이며, **논블로킹**(Non-blocking) **I/O**와 **단일 스레드** 이벤트 루프를 통한 높은 처리 성능을 가지고 있다.

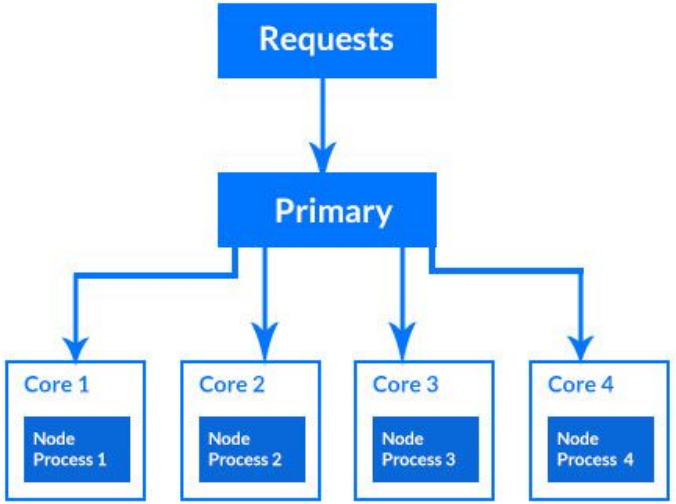
https://ko.wikipedia.org/wiki/Node.js

single thread, cpu가 논다!

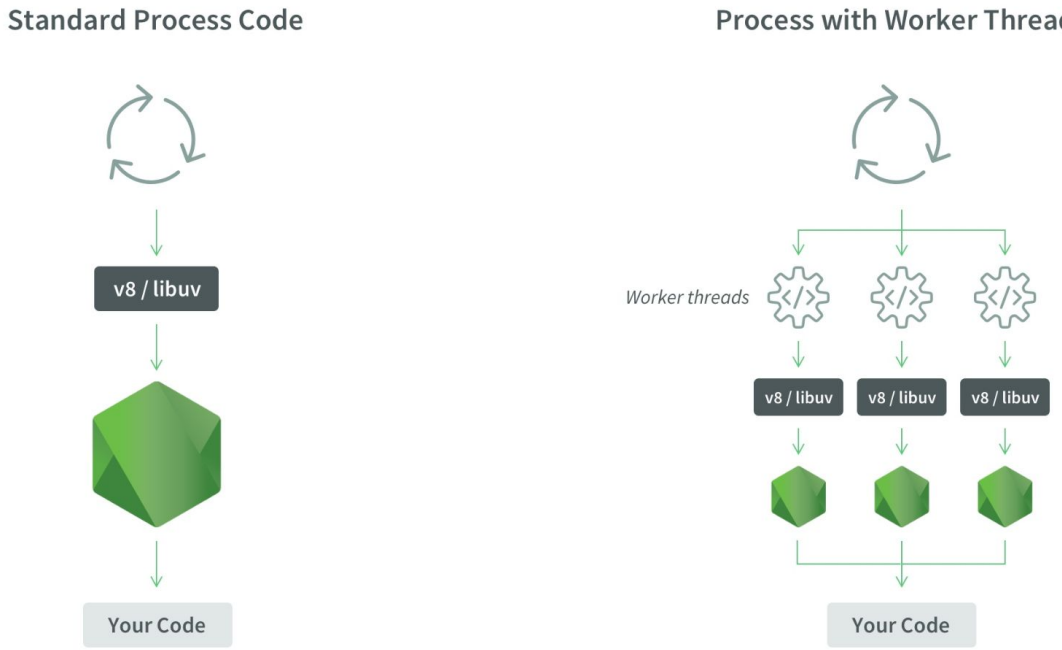


Node.js Cluster (multi-process)

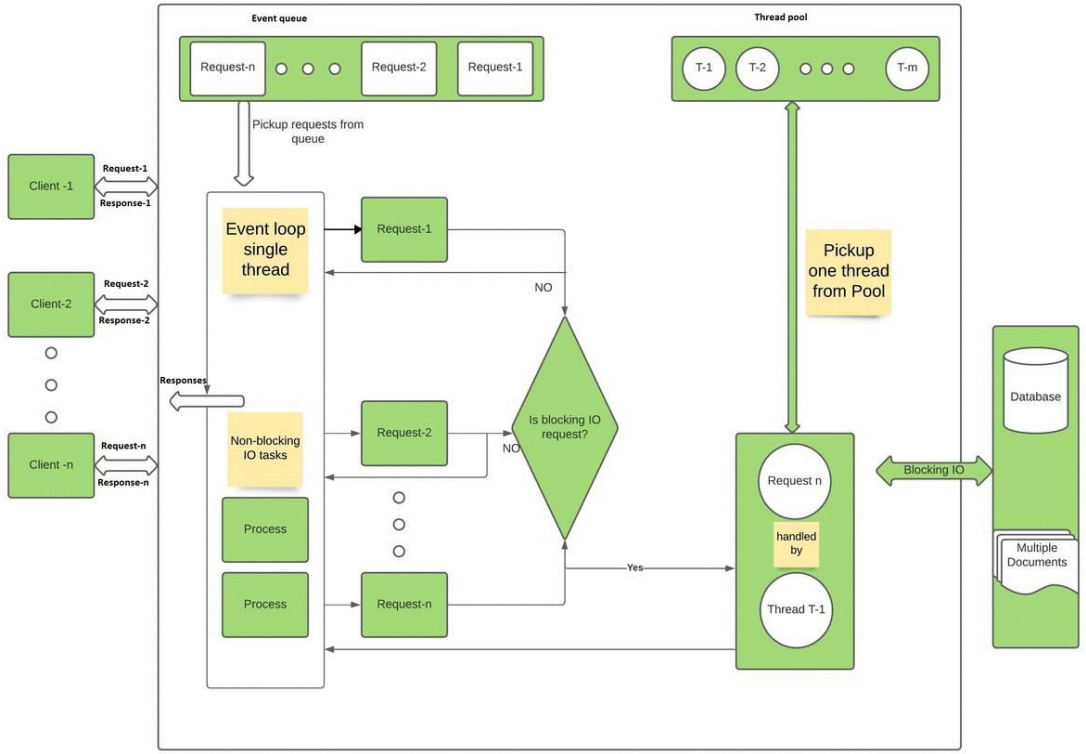




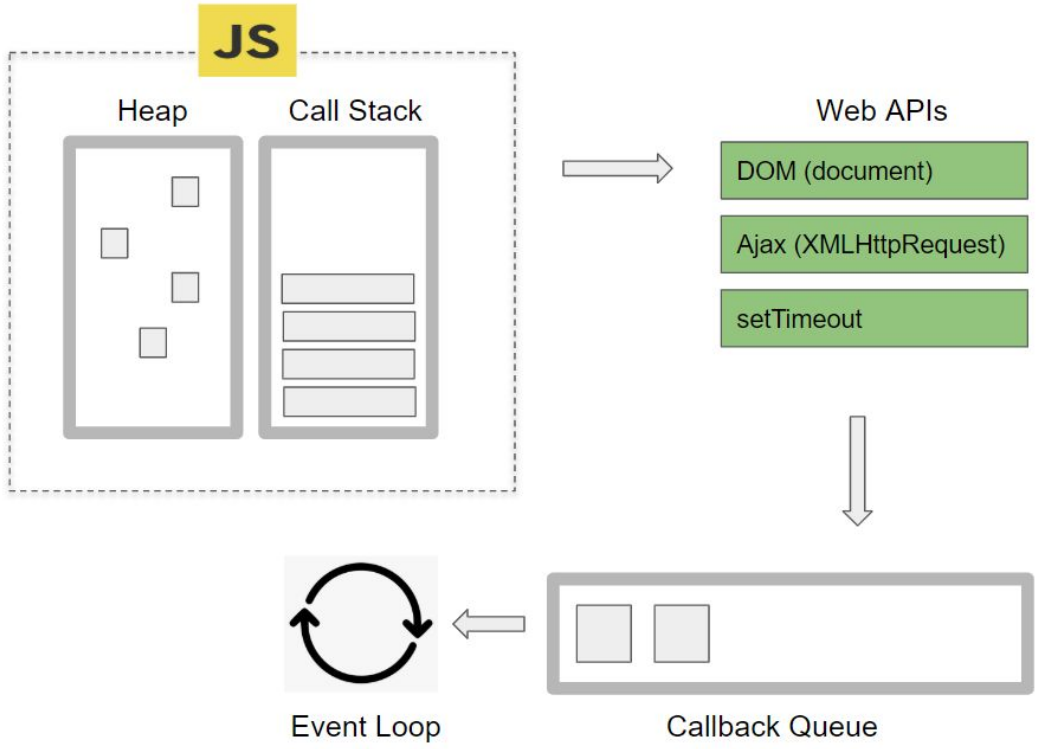
Node.js Worker Thread (multi-thread)



Node.js Thread pool

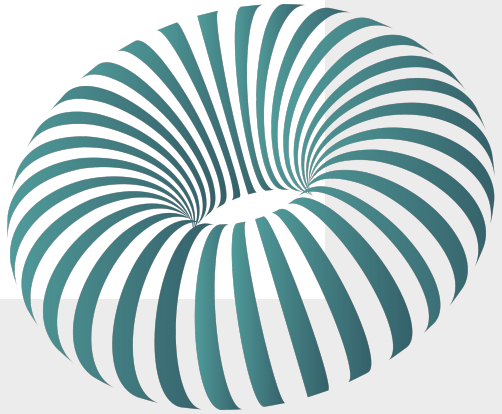


Node.js === single-thread?



Node.js의 Benchmark

(feat. cluster vs non-cluster)



```
// sync, blocking (cpu intensive)
function encryptUserData(userId, password, secretKey) {
    // 1. secretkey SHA-256 해시 생성
    const key = crypto
     .createHash("sha256")
     .update(String(secretKey))
     .digest("base64")
     .substring(0, 32);
    // 2. 초기화 벡터 생성
    const iv = crypto.randomBytes(16);
}
```

```
// sync, blocking (cpu intensive)
function encryptUserData(userId, password, secretKey) {
    ...
    // 3. USER ID + PASSWORD 암호화 (aes-256-cbc)
  const cipher = crypto.createCipheriv("aes-256-cbc", Buffer.from(key), iv);
  let encryptedUserId = cipher.update(userId, "utf8", "hex");
  encryptedUserId += cipher.final("hex");
  const cipherPassword = crypto.createCipheriv(
    "aes-256-cbc",
    Buffer.from(key),
    iv
  );
  let encryptedPassword = cipherPassword.update(password, "utf8", "hex");
  encryptedPassword += cipherPassword.final("hex");
}
```

Benchmark Info

Benchmark Tool, HEY

Environment

window

intel i5-12450H, 8 cores 12 threads, 16GB RAM

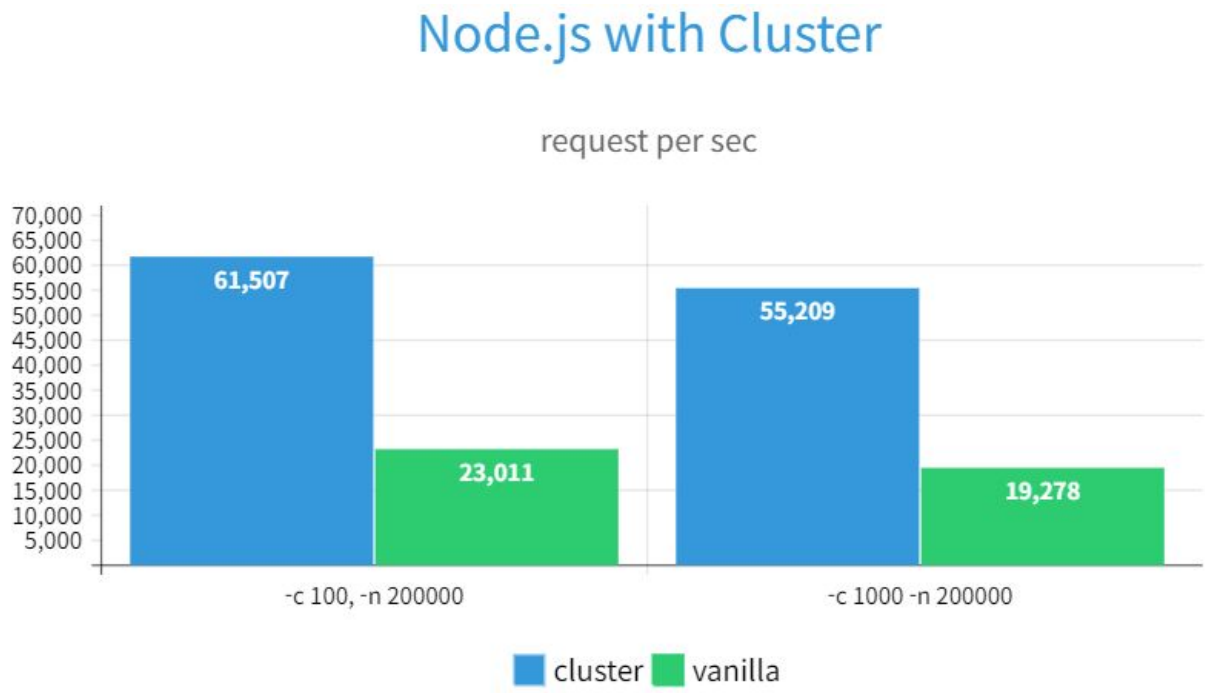
Benchmark Command

sync/blocking => hey -n 200000 -c 100|1000 http://localhost:3000

async/non-blocking => hey -n 1000 -c 10|100 http://localhost:3000

Benchmark Result (sync, blocking)

Request Per Sec (higher is better)



Benchmark Result (sync, blocking)

Error rate (lower is better)

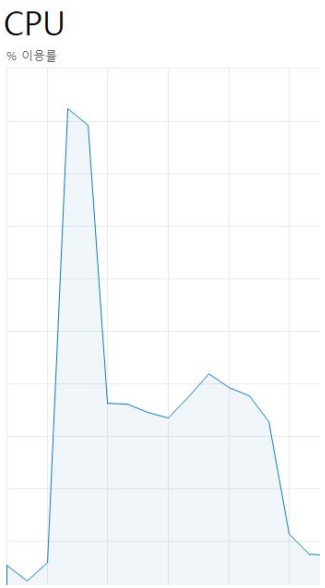
Node.js (vanilla) : -c 1000 => 6.3% (12600)

Node.js (cluster) : -c 1000 => 0.009% (18)

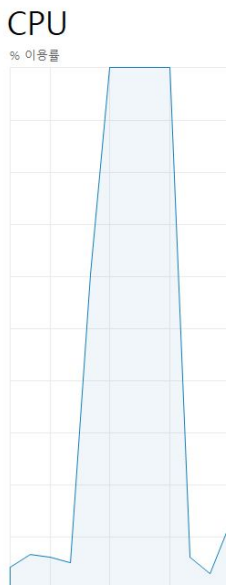
Benchmark Result (sync, blocking)

Cpu Usage

Vanilla



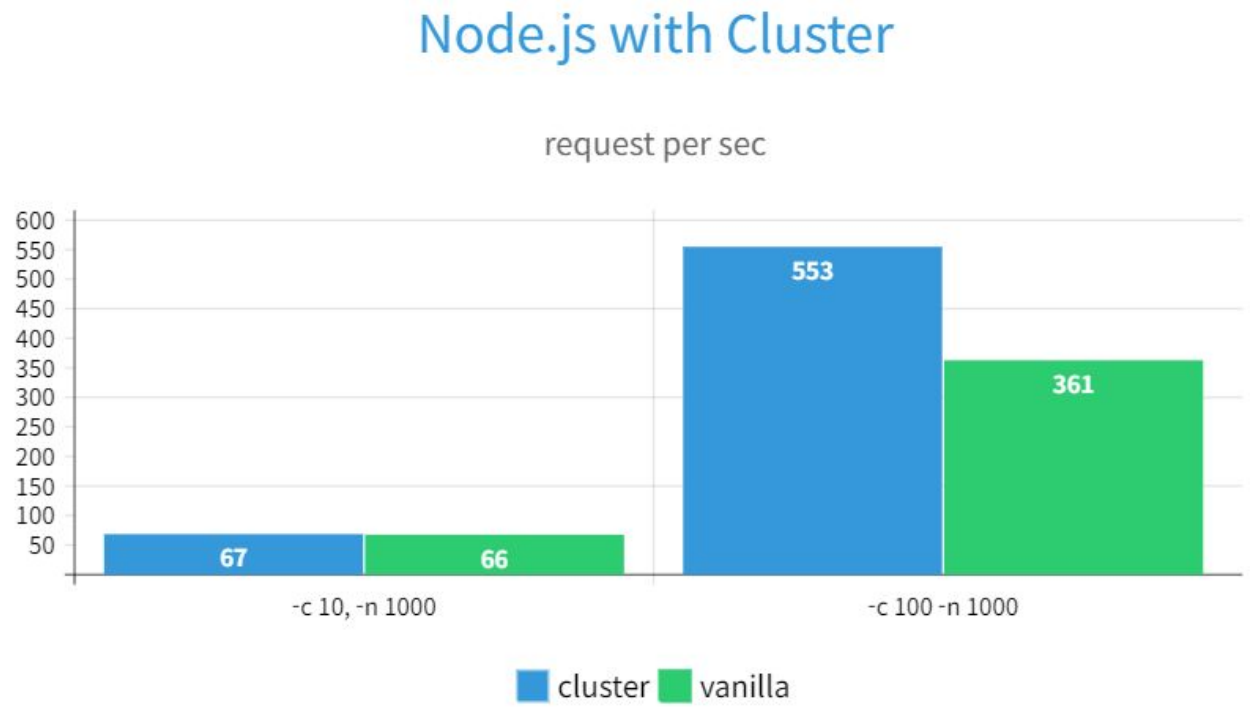
Cluster



```
// async, non-blocking (i/o intensive)
const requestListener = async (req, res) => {
    try {
      const response = await axios.get("http://example.com/");
      const htmlContent = response.data;
      res.writeHead(200, { "Content-Type": "text/html" });
      res.end(htmlContent);
    } catch (error) {
      console.error("Error fetching data from example.com:", error);
      res.writeHead(500, { "Content-Type": "text/plain" });
      res.end("Error fetching data");
    }
  }
};
```

Benchmark Result (async, non-blocking)

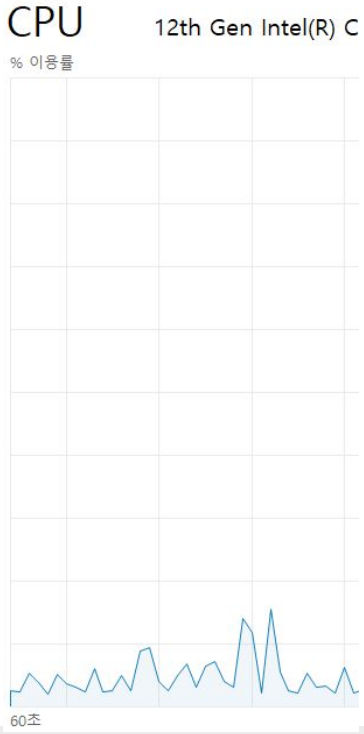
Request Per Sec (higher is better)



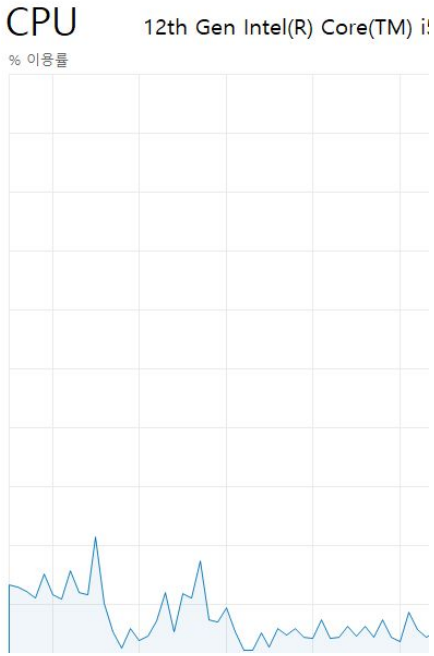
Benchmark Result (async, non-blocking)

Cpu Usage

Vanilla

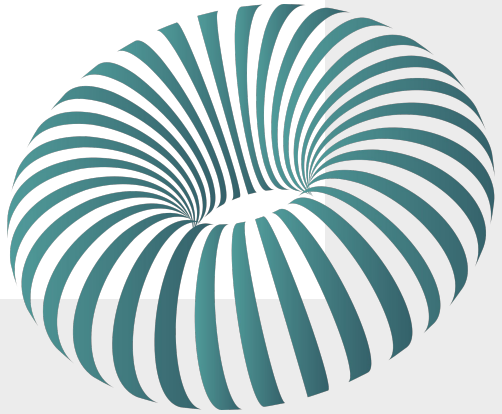


Cluster



Go Thread

(feat. goroutine)

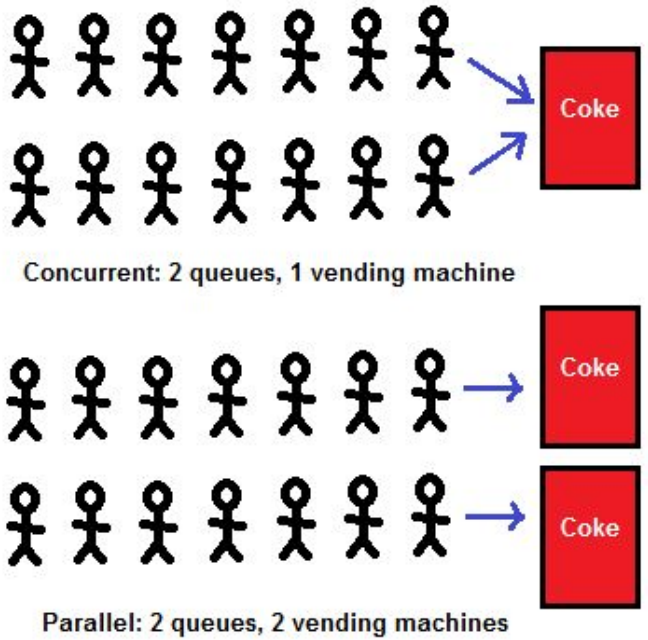


What is Go?

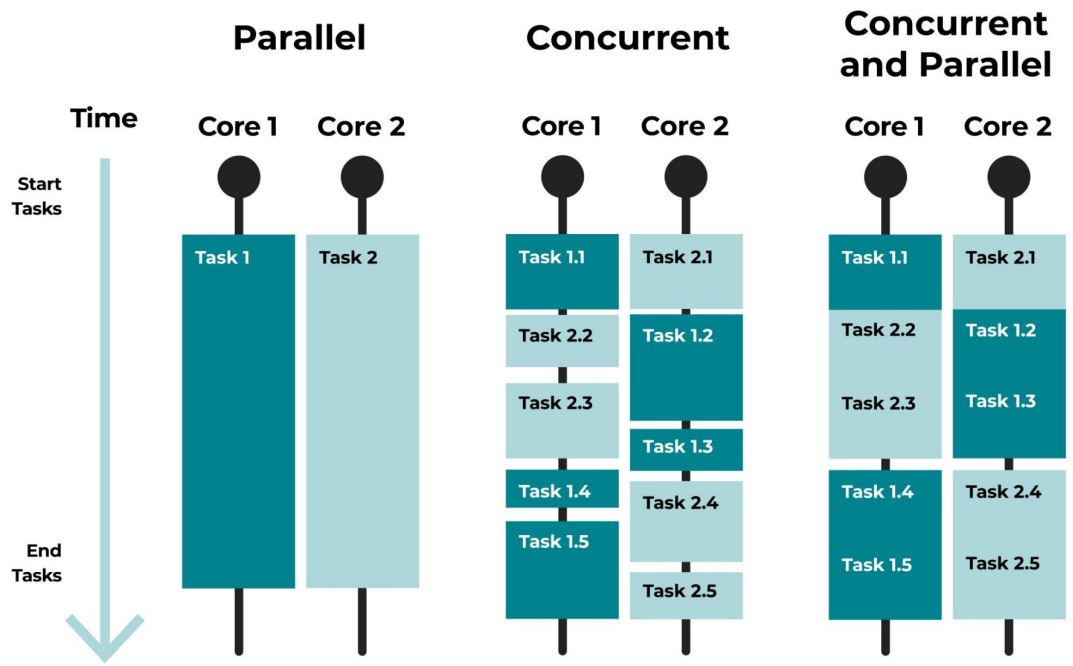
Go는 2009년 구글에서 일하는 로버트 그리즈머, 롭 파이크, 켄 톰프슨이 개발한 프로그래밍 언어이다. 가비지 컬렉션 기능이 있고, **병행성**(concurrent)을 잘 지원하는 컴파일 언어다.

https://ko.wikipedia.org/wiki/Go_(%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D_%EC%96%B8%EC%96%B4)

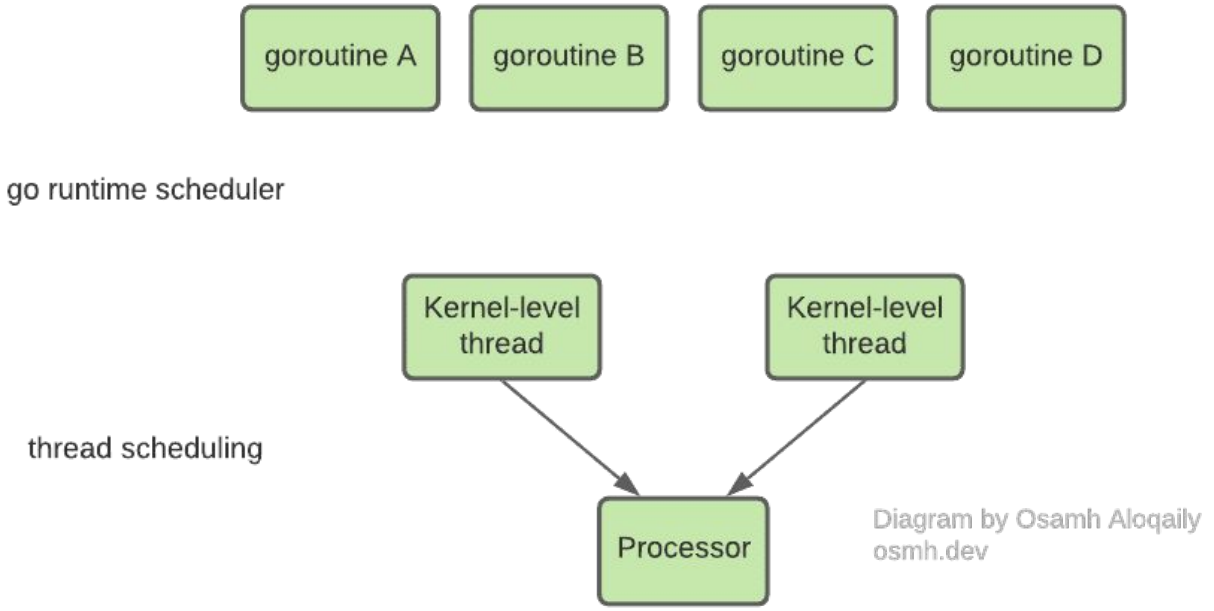
Concurrency is not parallelism



Concurrency is not parallelism

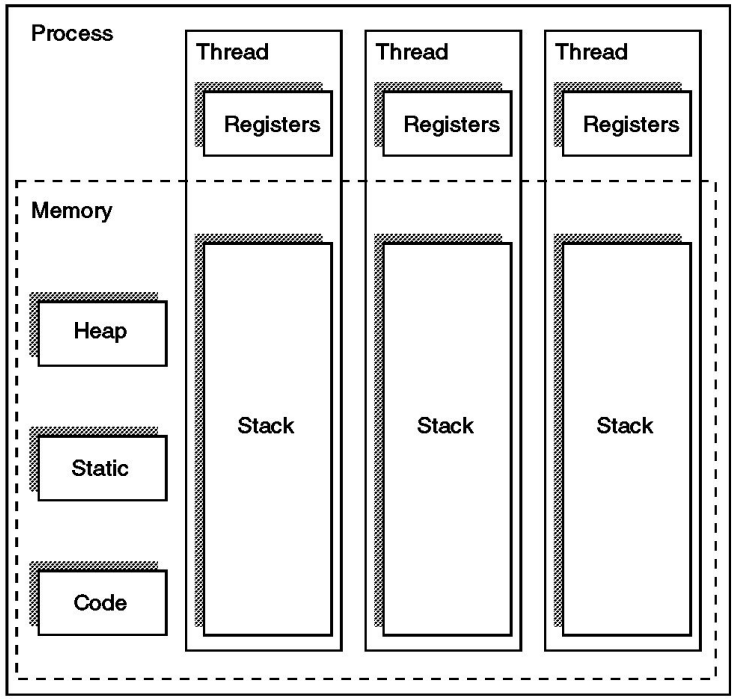


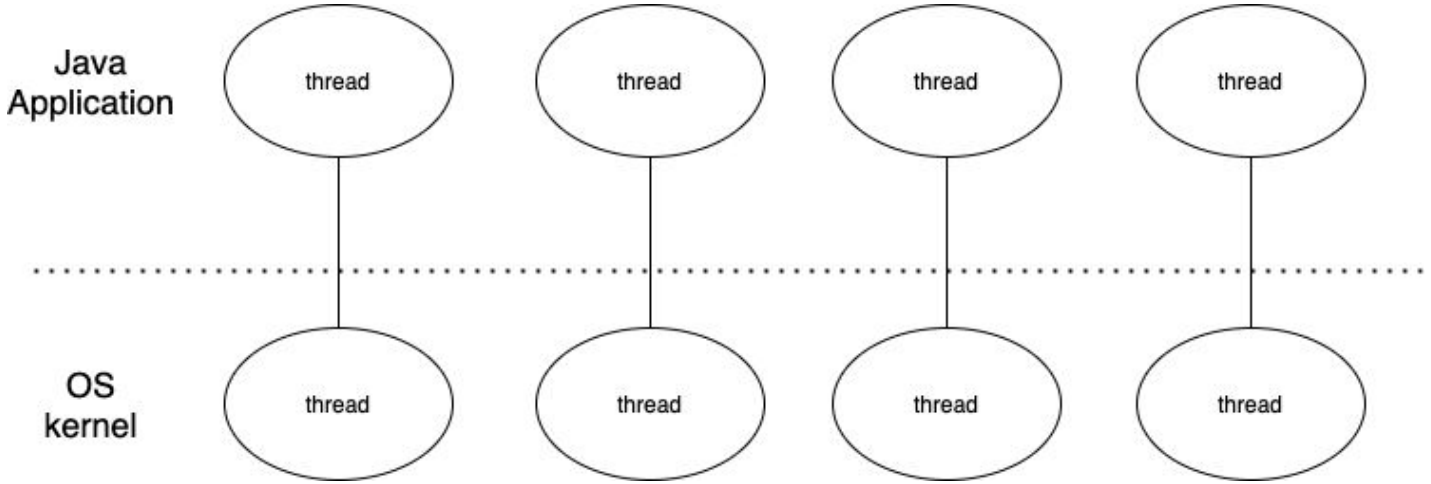
go routine


goroutine A
goroutine B
goroutine C
goroutine D
go runtime scheduler
Kernel-level thread
Kernel-level thread
thread scheduling
Processor
Diagram by Osamh Aloqaily
osmh.dev

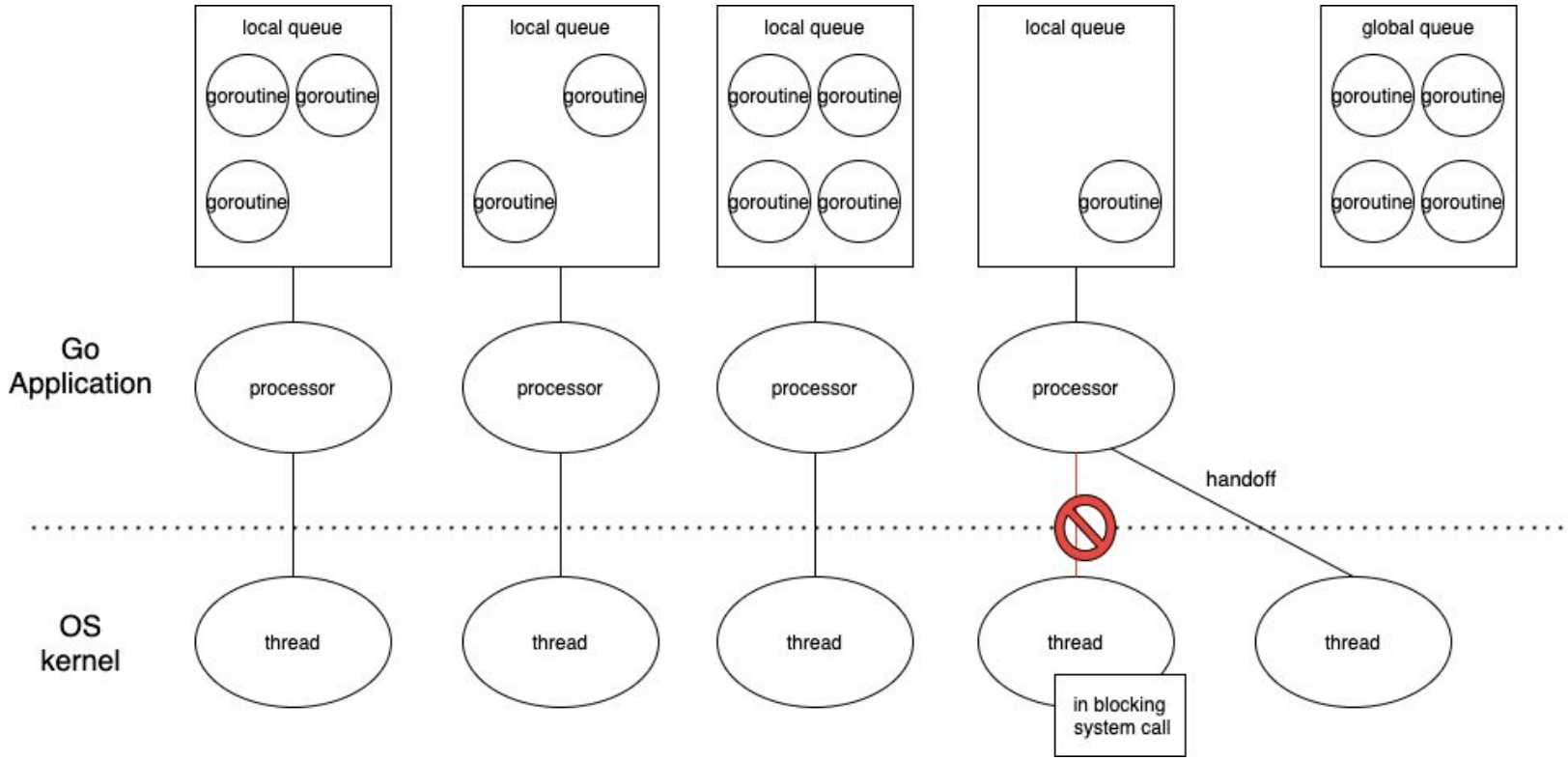
```go
package main
import "fmt"

func test() {
    fmt.Println("고루틴 test 함수 실행")
}

func main() {
    fmt.Println("main 함수 실행")
    go test() // super simple!
}
```

traditional multi-thread



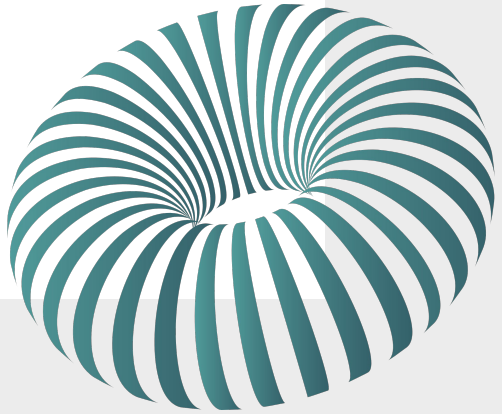


lightweight thread!



Benchmark Go

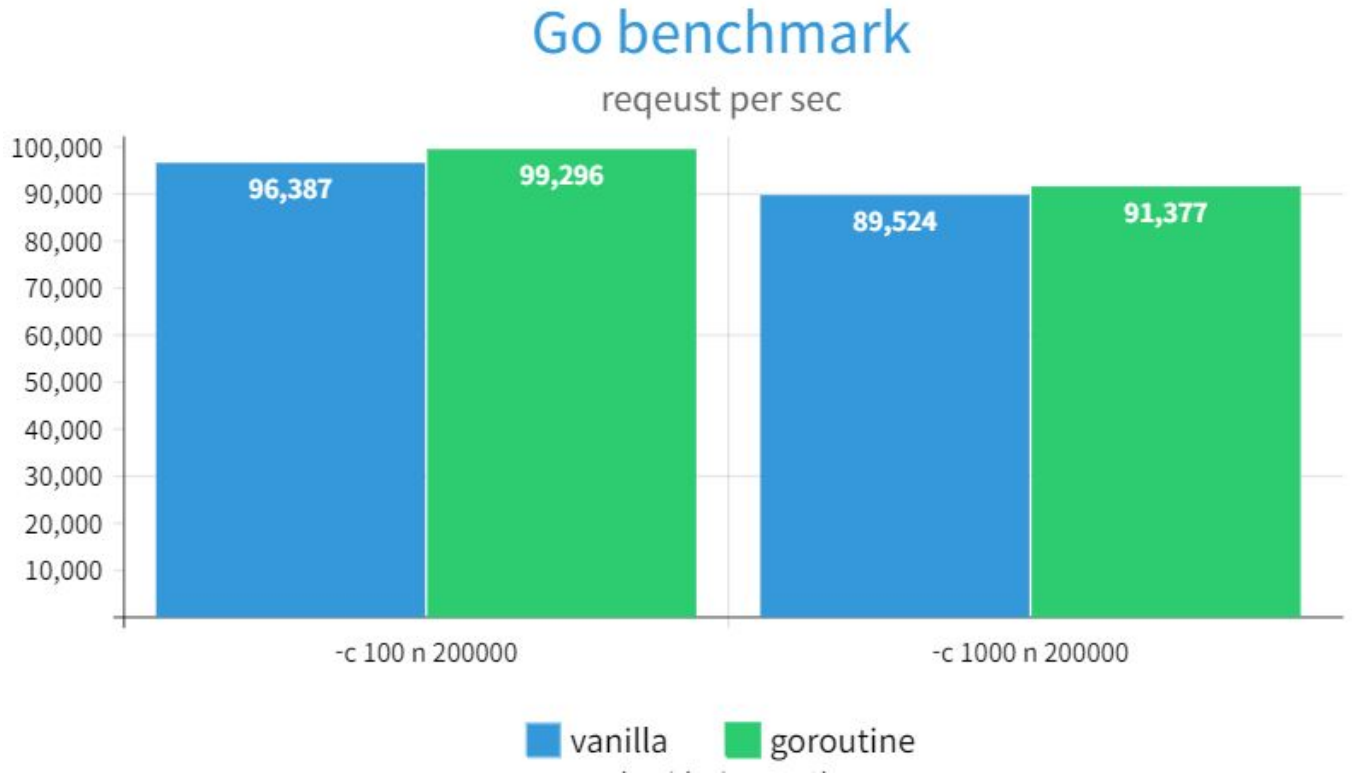
(feat. goroutine vs non-goroutine)



```go
// sync, blocking (cpu intensive)
func encryptUserData(userId, password, secretKey string) (map[string]string,
error) {
    // 1. secretkey SHA-256 해시 생성
    hasher := sha256.New()
    hasher.Write([]byte(secretKey))
    key := hasher.Sum(nil)[:32]
    block, err := aes.NewCipher(key)
    // 2. 초기화 벡터 생성
    iv := make([]byte, aes.BlockSize)
    if _, err := io.ReadFull(rand.Reader, iv); err != nil {
        return nil, err
    }
}
```

```go
// sync, blocking (cpu intensive)
func encryptUserData(userId, password, secretKey string) (map[string]string,
error) {
    ...
     // 3. USER ID + PASSWORD 암호화 (aes-256-cbc)
    cipherUserId := make([]byte, len(userId))
    cfbEncrypter := cipher.NewCFBEncrypter(block, iv)
    cfbEncrypter.XORKeyStream(cipherUserId, []byte(userId))

    cipherPassword := make([]byte, len(password))
    cfbEncrypter = cipher.NewCFBEncrypter(block, iv)
    cfbEncrypter.XORKeyStream(cipherPassword, []byte(password))
}
```

Benchmark Result (sync, blocking)



Benchmark Result (sync, blocking)

Error rate (lower is better)

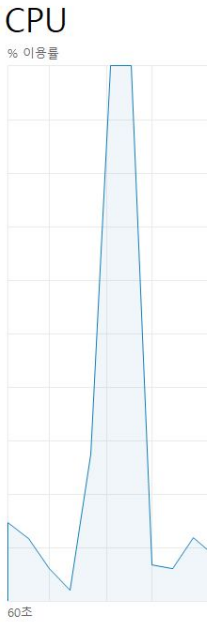
go (vanilla) : -c 1000 => 0.17% (351)

go (goroutine) : -c 1000 => 0.18% (361)

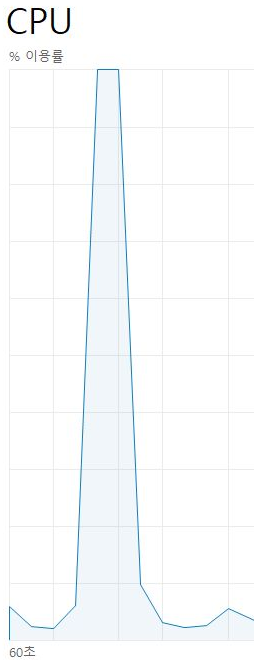
Benchmark Result (async, non-blocking)

Cpu Usage

Vanilla

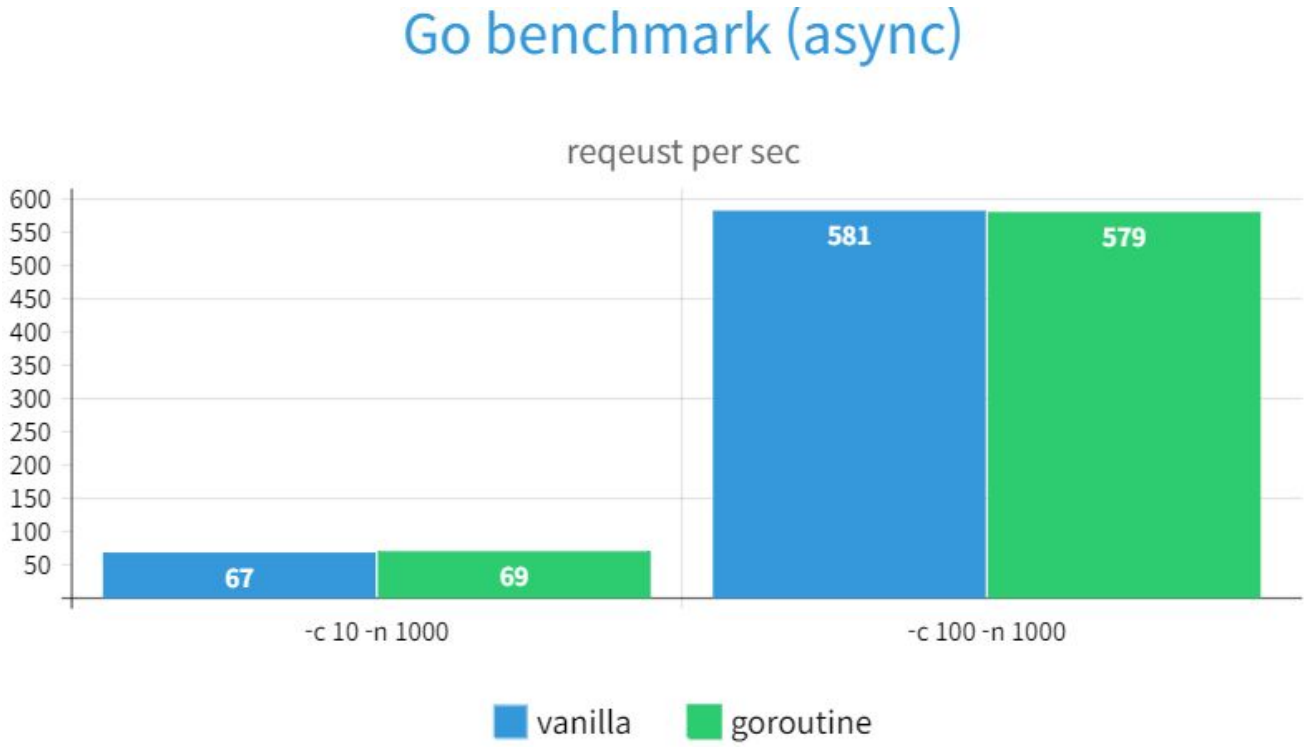


goroutine

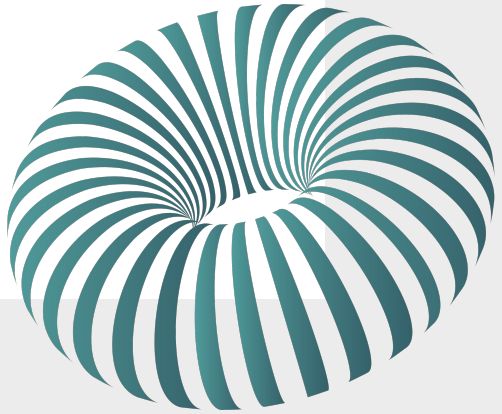


```go
// async, non-blocking (i/o intensive)
func requestHandler(w http.ResponseWriter, r *http.Request) {
    response, err := http.Get("http://example.com/")
    if err != nil {
        fmt.Println("Error fetching data:", err)
        http.Error(w, "Error fetching data", http.StatusInternalServerError)
        return
    }
    defer response.Body.Close()
    htmlContent, err := io.ReadAll(response.Body)
    w.WriteHeader(http.StatusOK)
    w.Header().Set("Content-Type", "text/html")
    w.Write(htmlContent)
}
```

Benchmark Result (async, non-blocking)

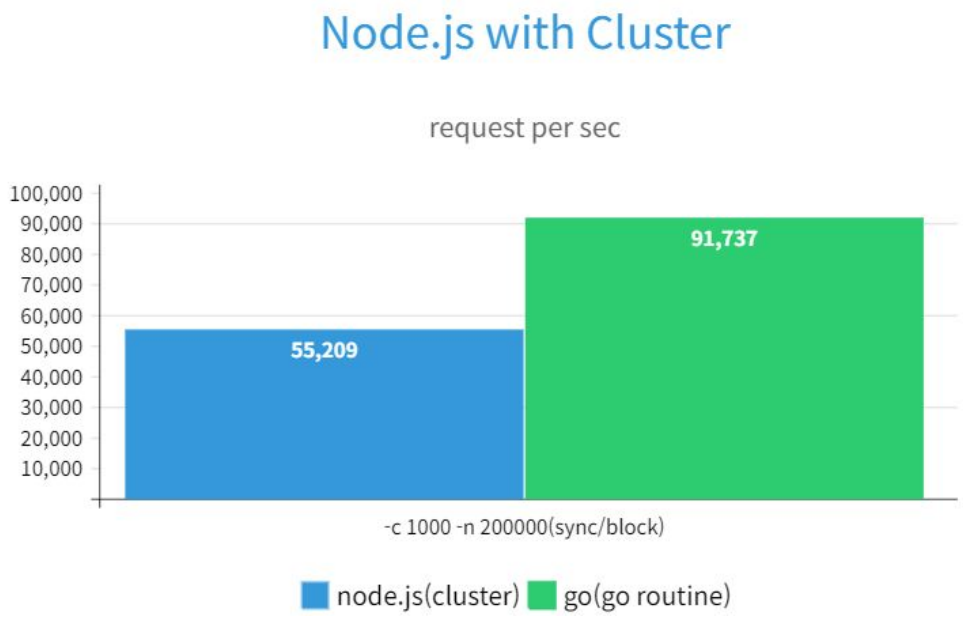


Result



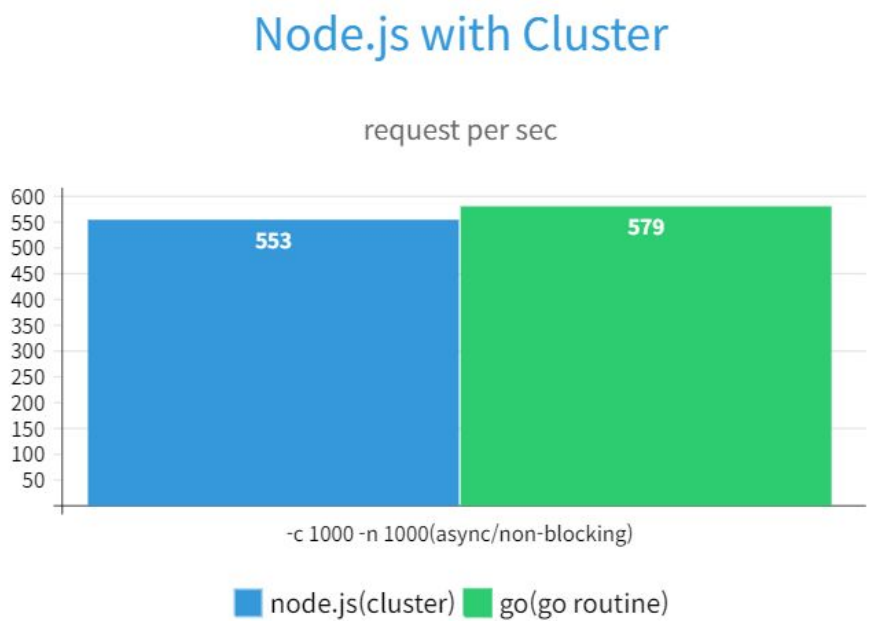
Result

The Go server outperformed a Node.js server with clustering by 66% in a cpu intensive-wise



Result

There was no difference between Node.js with cluster and Go in a I/O intensive-wise.



Insight

- Thanks to Node.js clustering, the error rate is slightly more stable compared to Go servers.
- For I/O-bound tasks, Node.js servers are sufficiently fast.
- For CPU-bound tasks with high concurrent requests, Go servers are significantly faster.

Q&A

Thank you!