Piotr Kołaczkowski

Home About Software

How Much Memory Do You Need to Run 1 Million Concurrent Tasks?

May 21, 2023

In this blog post, I delve into the comparison of memory consumption between asynchronous and multi-threaded programming across popular languages like Rust, Go, Java, C#, Python, Node.js and Elixir.

Some time ago I had to compare performance of a few computer programs designed to handle a large number of network connections. I saw huge differences in memory consumption of those programs, even exceeding 20x. Some programs consumed little over 100 MB, but the others reached almost 3 GB at 10k connections. Unfortunately those programs were quite complex and differed also in features, so it would be hard to compare them directly and draw some meaningful conclusions, as that wouldn't be an apple-to-apple comparison. This led me to an idea of creating a synthetic benchmark instead.

Benchmark

I created the following program in various programming languages:

> Let's launch N concurrent tasks, where each task waits for 10 seconds and then the program exists after all tasks finish. The number of tasks is controlled by the command line argument.

With a little help of ChatGPT I could write such program in a few minutes, even in programming languages I don't use every day. For your convenience, all benchmark code is available on my GitHub.

Rust

I created 3 programs in Rust. The first one uses traditional threads. Here is the core of it:

```
let mut handles = Vec::new();
for _ in 0..num_threads {
    let handle = thread::spawn(|| {
        thread::sleep(Duration::from_secs(10));
    });
    handles.push(handle);
}
for handle in handles {
    handle.join().unwrap();
}
```

The two other versions use async, one with `tokio` and the other one `async-std`. Here is the core of the `tokio` variant:

```
let mut tasks = Vec::new();
for _ in 0..num_tasks {
    tasks.push(task::spawn(async {
        time::sleep(Duration::from_secs(10)).await;
    }));
}
for task in tasks {
    task.await.unwrap();
}
```

The `async-std` variant is very similar, so I won't quote it here.

Go

In Go, goroutines are the building block for concurrency. We don't await them separately, but we use a `WaitGroup` instead:

```
var wg sync.WaitGroup
for i := 0; i < numRoutines; i++ {
    wg.Add(1)
    go func() {
        defer wg.Done()
        time.Sleep(10 * time.Second)
    }()
}
wg.Wait()
```

Java

Java traditionally uses threads, but JDK 21 offers a preview of virtual threads, which are a similar concept to goroutines. Therefore I created two variants of the benchmark. I was also curious how Java threads compare to Rust's threads.

```
List<Thread> threads = new ArrayList<>();
for (int i = 0; i < numTasks; i++) {
    Thread thread = new Thread(() -> {
        try {
            Thread.sleep(Duration.ofSeconds(10));
        } catch (InterruptedException e) {
        }
    });
    thread.start();
    threads.add(thread);
}
for (Thread thread : threads) {
    thread.join();
}
```

And here is the variant with virtual threads. Note how similar it is! Almost identical!

```
List<Thread> threads = new ArrayList<>();
for (int i = 0; i < numTasks; i++) {
    Thread thread = Thread.startVirtualThread(() -> {
        try {
            Thread.sleep(Duration.ofSeconds(10));
        } catch (InterruptedException e) {
        }
    });
    threads.add(thread);
}
for (Thread thread : threads) {
    thread.join();
}
```

C#

C#, similar to Rust, has first-class support for async/await:

```
List<Task> tasks = new List<Task>();
for (int i = 0; i < numTasks; i++)
{
    Task task = Task.Run(async () =>
    {
        await Task.Delay(TimeSpan.FromSeconds(10));
    });
    tasks.Add(task);
}
await Task.WhenAll(tasks);
```

Node.JS

So does Node.JS:

```
const delay = util.promisify(setTimeout);
const tasks = [];

for (let i = 0; i < numTasks; i++) {
    tasks.push(delay(10000));
}

await Promise.all(tasks);
```

Python

And Python added async/await in 3.5, so we can write:

```
async def perform_task():
    await asyncio.sleep(10)


tasks = []

for task_id in range(num_tasks):
    task = asyncio.create_task(perform_task())
    tasks.append(task)

await asyncio.gather(*tasks)
```

Elixir

Elixir is famous for its async capabilities as well:

```
tasks =
    for _ <- 1..num_tasks do
        Task.async(fn ->
            :timer.sleep(10000)
        end)
    end

Task.await_many(tasks, :infinity)
```

Test Environment

- Hardware: Intel(R) Xeon(R) CPU E3-1505M v6 @ 3.00GHz
- OS: Ubuntu 22.04 LTS, Linux p5520 5.15.0-72-generic
- Rust: 1.69
- Go: 1.18.1
- Java: OpenJDK “21-ea” build 21-ea+22-1890
- .NET: 6.0.116
- Node.JS: v12.22.9
- Python: 3.10.6
- Elixir: Erlang/OTP 24 erts-12.2.1, Elixir 1.12.2

All programs were launched using the release mode if available. Other options were left default.

Results

Minimum Footprint

Let’s start from something small. Because some runtimes require some memory for themselves, let’s first launch only one task.

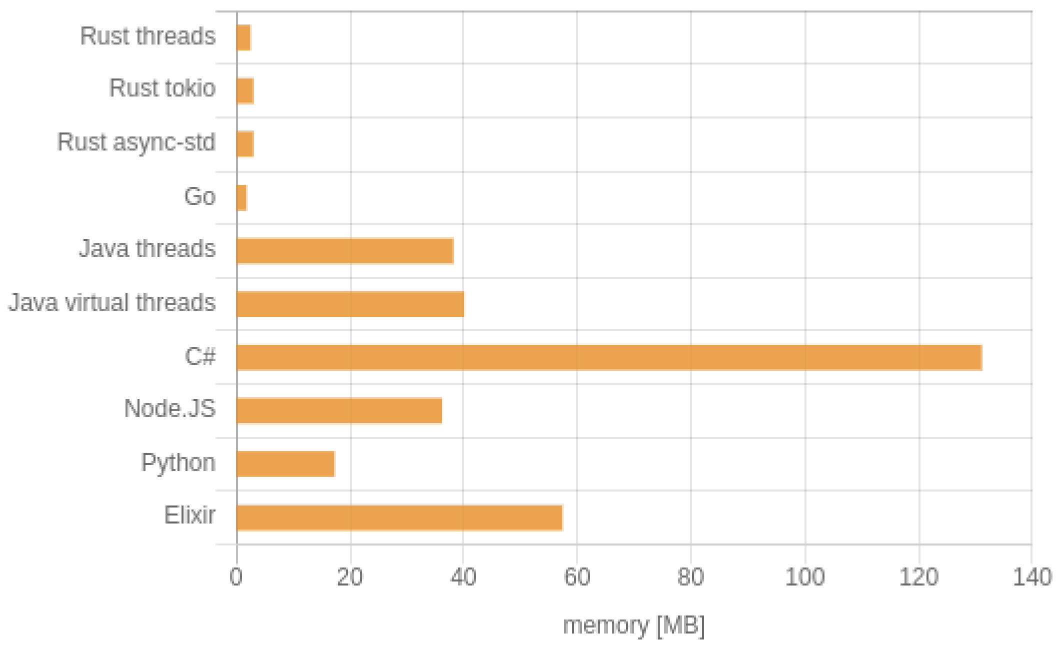

Fig.1: Peak memory needed to launch one task

We can see there are certainly two groups of programs. Go and Rust programs, compiled statically to native binaries, need very little memory. The other programs running on managed platforms or through interpreters consume more memory, although Python fares really well in this case. There is about an order of magnitude difference in memory consumption between those two groups.

It is a surprise to me that .NET somehow has the worst footprint, but I guess this can be tuned with some settings maybe. Let me know in the comments if there are any tricks. I haven’t seen much difference between the debug and release modes.

10k Tasks

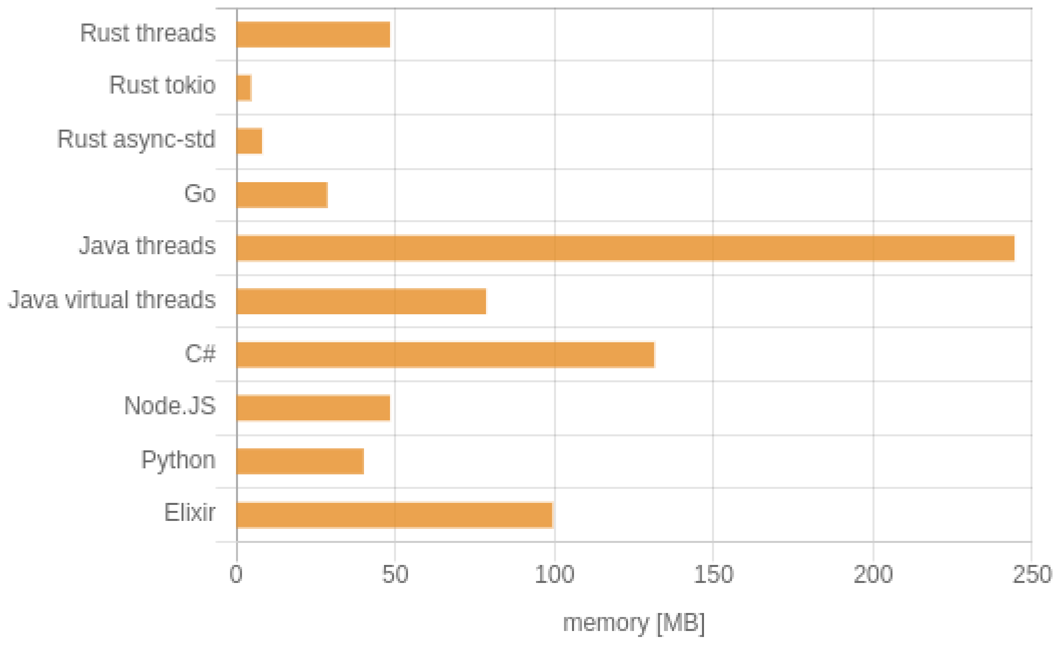

Fig.2: Peak memory needed to launch 10,000 tasks

A few surprises here! Everybody probably expected that threads would be a big loser of this benchmark. And this is true for Java threads, which indeed consumed almost 250 MB of RAM. But native Linux threads used from Rust seem to be lightweight enough that at 10k threads the memory consumption is still lower than the idle memory consumption of many other runtimes. Async tasks or virtual (green) threads might be lighter than native threads, but we won't see that advantage at only 10k tasks. We need more tasks.

Another surprise here is Go. Goroutines are supposed to be very lightweight, but they actually consumed more than 50% of RAM required by Rust threads. Honestly, I was expecting much bigger difference in favor of Go. Hence, I conclude that at 10k concurrent tasks, threads are still quite a competitive alternative. Linux kernel definitely does something right here.

Go has also lost its tiny advantage it had over Rust async in the previous benchmark and now it consumes over 6x more memory than the best Rust program. It was also overtaken by Python.

And the final surprise is that at 10k tasks the memory consumption of .NET didn't significantly go up from the idle memory use. Probably it just uses preallocated memory. Or its idle memory use is so high that 10k tasks is just too few to matter.

100k Tasks

I could not launch 100,000 threads on my system, so the threads benchmarks had to be excluded. Probably this could be somehow tweaked byt changing system settings, but after trying for an hour I gave up. So at 100k tasks you probably don't want to use threads.

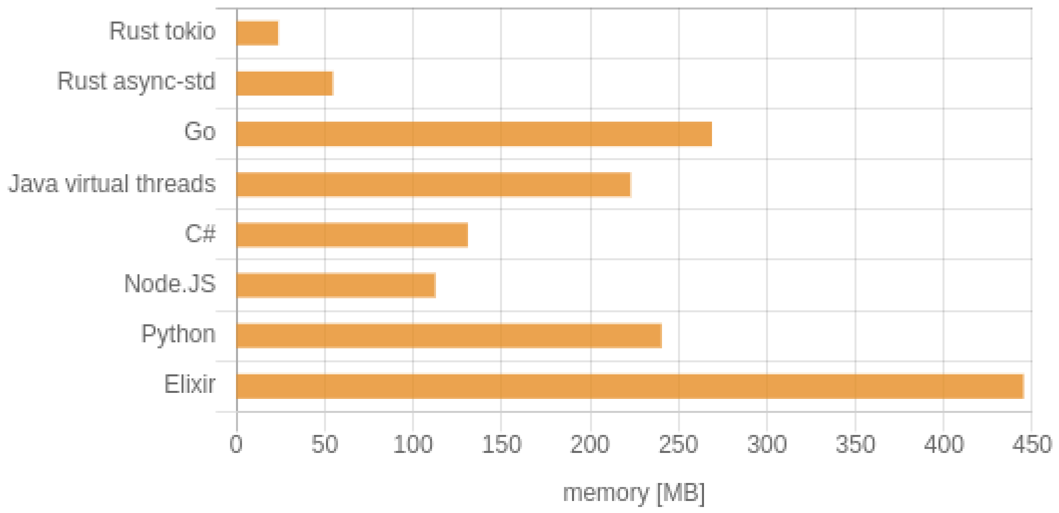

Fig.3: Peak memory needed to launch 100,000 tasks

At this point the Go program has been beaten up not only by Rust, but also by Java, C# and Node.JS.

And Linux .NET likely cheats because its memory use still didn't go up. ;) I had to double check if it really launches the right number of tasks, but indeed, it does. And it still exits after about 10 seconds, so it doesn't block the main loop. Magic! Good job, .NET.

1 Million Tasks

Let's go extreme now.

At 1 million tasks, Elixir gave up with `** (SystemLimitError) a system limit has been reached`. Edit: Some commenters pointed out I could increase the process limit. After adding `--erl '+P 1000000'` parameter to `elixir` invocation, it ran fine.

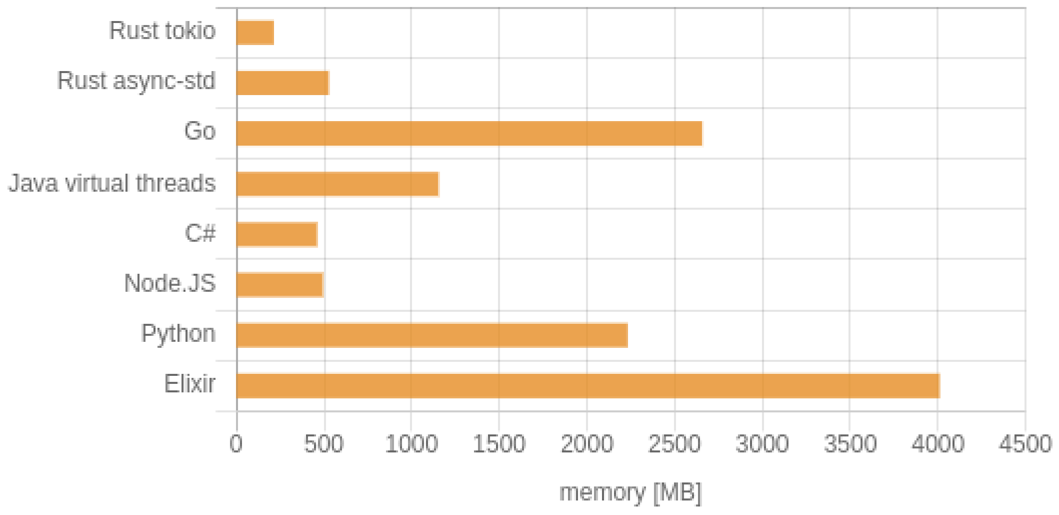

Fig.4: Peak memory needed to launch 1 million tasks

Finally we see the increase in memory consumption of a C# program. But it is still very competitive. It even managed to slightly beat one of the Rust runtimes!

The distance between Go and others increased. Now Go loses over 12x to the winner. It also loses over 2x to Java, which contradicts the general perception of JVM being a memory hog and Go being lightweight.

Rust `tokio` remained unbeatable. This isn't surprising after seeing how it did at 100k tasks.

Final Word

As we have observed, a high number of concurrent tasks can consume a significant amount of memory, even if they do not perform complex operations. Different language runtimes have varying trade-offs, with some being lightweight and efficient for a small number of tasks but scaling poorly with hundreds of thousands of tasks. Conversely, other runtimes with high initial overhead can handle high workloads effortlessly. It is important to note that not all runtimes were even capable of handling a very large number of concurrent tasks with default settings.

This comparison focused solely on memory consumption, while other factors such as task launch time and communication speed are equally important. Notably, at 1 million tasks, I observed that the overhead of launching tasks became evident, and most programs required more than 12 seconds to complete. Stay tuned for upcoming benchmarks, where I will explore additional aspects in depth.

Share on:

Sponsored

Teacher Proves Experts Wrong - Puts His Money Where Their Mouth Is
Spoutlofi

Newcastle Homeowners Can Now Go Solar For Almost No-Cost
Australian Solar Rebates

Free Course: How To Trade US Companies From Australia
Trading Mastery

Download Cryptocurrency Income Report
ASRW

Australians Over 50 Can Try Mini Hearing Aids For Free
Try High Tech Hearing Devices

Wanted: Australians who want to try Latest High-Tech Hearing Aids - Free!
Auditorey.com | Try Latest Hearing Devices
Try Now

Previous Post
Don't Share Java FileChannels