Debugging performance issues in Go programs

-- originally written by Dmitry Vyukov

Let's assume you have a Go program and want to improve its performance. There are several tools available that can help with this task. These tools can help you to identify various types of hotspots (CPU, IO, memory), hotspots are the places that you need to concentrate on in order to significantly improve performance. However, another outcome is possible -- the tools can help you identify obvious performance defects in the program. For example, you prepare an SQL statement before each query while you could prepare it once at program startup. Another example is if an O(N^2) algorithm somehow slipped into where an obvious O(N) exists and is expected. In order to identify such cases you need to sanity check what you see in profiles. For example for the first case significant time spent in SQL statement preparation would be the red flag.

It's also important to understand various bounding factors for performance. For example, if the program communicates via 100 Mbps network link and it is already utilizes >90Mbps, there is not much you can do with the program to improve its performance. There are similar bounding factors for disk IO, memory consumption and computational tasks. With that in mind we can look at the available tools.

Note: The tools can interfere with each other. For example, precise memory profiling skews CPU profiles, goroutine blocking profiling affects scheduler trace, etc. Use tools in isolation to get more precise info.

CPU Profiler

Go runtime contains built-in CPU profiler, which shows what functions consume what percent of CPU time. There are 3 ways you can get access to it:

1. The simplest one is -cpuprofile flag of 'go test' (https://pkg.go.dev/cmd/go/#hdr-Description_of_testing_flags) command. For example, the following command:

   ```
   $ go test -run=none -bench=ClientServerParallel4 -cpuprofile=cprof net/http
   ```

   will profile the given benchmark and write CPU profile into 'cprof' file. Then:

   ```
   $ go tool pprof --text http.test cprof
   ```

   will print a list of the hottest functions.

   There are several output types available, the most useful ones are: `--text`, `--web` and `--list`. Run `go tool pprof` to get the complete list. The obvious drawback of this option is that it works only for tests.

2. net/http/pprof: This is the ideal solution for network servers. You merely need to import `net/http/pprof`, and collect profiles with:

   ```
   $ go tool pprof --text mybin  http://myserver:6060:/debug/pprof/profile
   ```

3. Manual profile collection. You need to import runtime/pprof and add the following code to main function:

   ```
   if *flagCpuprofile != "" {
       f, err := os.Create(*flagCpuprofile)
       if err != nil {
           log.Fatal(err)
       }
   ```

```
    pprof.StartCPUProfile(f)
    defer pprof.StopCPUProfile()
}
```

The profile will be written to the specified file, visualize it the same way as in the first option.

Here is an example of a profile visualized with `--web` option: [cpu_profile.png]

You can investigate a single function with `--list=funcname` option. For example, the following profile shows that the time was spent in the `append` function:

```
.      .   93: func (bp *buffer) WriteRune(r rune) error {
.      .   94:     if r < utf8.RuneSelf {
5      5   95:         *bp = append(*bp, byte(r))
.      .   96:         return nil
.      .   97:     }
.      .   98:
.      .   99:     b := *bp
.      .  100:     n := len(b)
.      .  101:     for n+utf8.UTFMax > cap(b) {
.      .  102:         b = append(b, 0)
.      .  103:     }
.      .  104:     w := utf8.EncodeRune(b[n:n+utf8.UTFMax], r)
.      .  105:     *bp = b[:n+w]
.      .  106:     return nil
.      .  107: }
```

There are also 3 special entries that the profiler uses when it can't unwind stack: GC, System and ExternalCode. GC means time spent during garbage collection, see Memory Profiler and Garbage Collector Trace sections below for optimization suggestions. System means time spent in goroutine scheduler, stack management code and other auxiliary runtime code. ExternalCode means time spent in native dynamic libraries.

Here are some hints with respect to how to interpret what you see in the profile.

If you see lots of time spent in `runtime.mallocgc` function, the program potentially makes excessive amount of small memory allocations. The profile will tell you where the allocations are coming from. See the memory profiler section for suggestions on how to optimize this case.

If lots of time is spent in channel operations, sync.Mutex code and other synchronization primitives or System component, the program probably suffers from contention. Consider to restructure program to eliminate frequently accessed shared resources. Common techniques for this include sharding/partitioning, local buffering/batching and copy-on-write technique.

If lots of time is spent in `syscall.Read/Write`, the program potentially makes excessive amount of small reads and writes. Bufio wrappers around os.File or net.Conn can help in this case.

If lots of time is spent in GC component, the program either allocates too many transient objects or heap size is very small so garbage collections happen too frequently. See Garbage Collector Tracer and Memory Profiler sections for optimization suggestions.

Note: For darwin CPU profiler currently only works on El Capitan or newer.

Note: On windows you need to install Cygwin, Perl and Graphviz to generate svg/web profiles.

Memory Profiler

Memory profiler shows what functions allocate heap memory. You can collect it in similar ways as CPU profile: with `go test --memprofile` (https://pkg.go.dev/cmd/go/#hdr-Description_of_testing_flags), with net/http/pprof via http://myserver:6060:/debug/pprof/heap or by calling runtime/pprof.WriteHeapProfile.

You can visualize only allocations live at the time of profile collection (`--inuse_space flag to pprof` , default), or all allocations happened since program start (`--alloc_space` flag to `pprof`). The former is useful for profiles collected with net/http/pprof on live applications, the latter is useful for profiles collected at program end (otherwise you will see almost empty profile).

Note: the memory profiler is sampling, that is, it collects information only about some subset of memory allocations. Probability of sampling an object is proportional to its size. You can change the sampling rate with go test `--memprofilerate` flag, or by setting `runtime.MemProfileRate` variable at program startup. The rate of 1 will lead to collection of information about all allocations, but it can slow down execution. The default sampling rate is 1 sample per 512KB of allocated memory.

You can also visualize number of bytes allocated or number of objects allocated (`--inuse/alloc_space` and `--inuse/alloc_objects` flags, respectively). The profiler tends to sample larger objects during profiling more. But it's important to understand that large objects affect memory consumption and GC time, while large number of tiny allocations affects execution speed (and GC time to some degree as well). So it may be useful to look at both.

Objects can be persistent or transient. If you have several large persistent objects allocated at program start, they will be most likely sampled by the profiler (because they are large). Such objects do affect memory consumption and GC time, but they do not affect normal execution speed (no memory management operations happen on them). On the other hand if you have large number of objects with very short life durations, they can be barely represented in the profile (if you use the default `--inuse_space mode`). But they do significantly affect execution speed, because they are constantly allocated and freed. So, once again, it may be useful to look at both types of objects. So, generally, if you want to reduce memory consumption, you need to look at `--inuse_space` profile collected during normal program operation. If you want to improve execution speed, look at `--alloc_objects` profile collected after significant running time or at program end.

There are several flags that control reporting granularity. `--functions` makes pprof report on function level (default). `--lines` makes pprof report on source line level, which is useful if hot functions allocate on different lines. And there are also `--addresses` and `--files` for exact instruction address and file level, respectively.

There is a useful option for the memory profile -- you can look at it right in the browser (provided that you imported `net/http/pprof`). If you open http://myserver:6060/debug/pprof/heap?debug=1 you must see the heap profile along the lines of:

```
heap profile: 4: 266528 [123: 11284472] @ heap/1048576
1: 262144 [4: 376832] @ 0x28d9f 0x2a201 0x2a28a 0x2624d 0x26188 0x94ca3 0x94a0b
0x17add6 0x17ae9f 0x1069d3 0xfe911 0xf0a3e 0xf0d22 0x21a70
#   0x2a201 cnew+0xc1   runtime/malloc.goc:718
#   0x2a28a runtime.cnewarray+0x3a          runtime/malloc.goc:731
#   0x2624d makeslice1+0x4d             runtime/slice.c:57
#   0x26188 runtime.makeslice+0x98          runtime/slice.c:38
#   0x94ca3 bytes.makeSlice+0x63            bytes/buffer.go:191
#   0x94a0b bytes.(*Buffer).ReadFrom+0xcb       bytes/buffer.go:163
#   0x17add6    io/ioutil.readAll+0x156         io/ioutil/ioutil.go:32
#   0x17ae9f    io/ioutil.ReadAll+0x3f          io/ioutil/ioutil.go:41
```

```
#	0x1069d3	godoc/vfs.ReadFile+0x133	    godoc/vfs/vfs.go:44
#	0xfe911	godoc.func·023+0x471	        godoc/meta.go:80
#	0xf0a3e	godoc.(*Corpus).updateMetadata+0x9e     godoc/meta.go:101
#	0xf0d22	godoc.(*Corpus).refreshMetadataLoop+0x42    godoc/meta.go:141

2: 4096 [2: 4096] @ 0x28d9f 0x29059 0x1d252 0x1d450 0x106993 0xf1225 0xe1489 0xfbcad
0x21a70
#	0x1d252	newdefer+0x112	          runtime/panic.c:49
#	0x1d450	runtime.deferproc+0x10	      runtime/panic.c:132
#	0x106993	godoc/vfs.ReadFile+0xf3	        godoc/vfs/vfs.go:43
#	0xf1225	godoc.(*Corpus).parseFile+0x75      godoc/parser.go:20
#	0xe1489	godoc.(*treeBuilder).newDirTree+0x8e9   godoc/dirtrees.go:108
#	0xfbcad	godoc.func·002+0x15d	        godoc/dirtrees.go:100
```

The numbers in the beginning of each entry `("1: 262144 [4: 376832]")` represent number of currently live objects, amount of memory occupied by live objects, total number of allocations and amount of memory occupied by all allocations, respectively.

Optimizations are usually application-specific, but here are some common suggestions.

1. Combine objects into larger objects. For example, replace `*bytes.Buffer` struct member with `bytes.Buffer` (you can preallocate buffer for writing by calling `bytes.Buffer.Grow` later). This will reduce number of memory allocations (faster) and also reduce pressure on garbage collector (faster garbage collections).

2. Local variables that escape from their declaration scope get promoted into heap allocations. Compiler generally can't prove that several variables have the same life time, so it allocates each such variable separately. So you can use the above advise for local variables as well. For example, replace:

   ```
   for k, v := range m {
       k, v := k, v   // copy for capturing by the goroutine
       go func() {
           // use k and v
       }()
   }
   ```

   with:

   ```
   for k, v := range m {
       x := struct{ k, v string }{k, v}   // copy for capturing by the goroutine
       go func() {
           // use x.k and x.v
       }()
   }
   ```

   This replaces two memory allocations with a single allocation. However, this optimization usually negatively affects code readability, so use it reasonably.

3. A special case of allocation combining is slice array preallocation. If you know a typical size of the slice, you can preallocate a backing array for it as follows:

   ```
   type X struct {
       buf      []byte
   ```

```
    bufArray [16]byte // Buf usually does not grow beyond 16 bytes.
}

func MakeX() *X {
    x := &X{}
    // Preinitialize buf with the backing array.
    x.buf = x.bufArray[:0]
    return x
}
```

4. If possible use smaller data types. For example, use `int8` instead of `int`.

5. Objects that do not contain any pointers (note that strings, slices, maps and chans contain implicit pointers), are not scanned by garbage collector. For example, a 1GB byte slice virtually does not affect garbage collection time. So if you remove pointers from actively used objects, it can positively impact garbage collection time. Some possibilities are: replace pointers with indices, split object into two parts one of which does not contain pointers.

6. Use freelists to reuse transient objects and reduce number of allocations. Standard library contains sync.Pool type that allows to reuse the same object several times in between garbage collections. However, be aware that, as any manual memory management scheme, incorrect use of sync.Pool can lead to use-after-free bugs.

You can also use the Garbage Collector Trace (see below) to get some insights into memory issues.

TODO(dvyukov): mention that stats are updated in a deferred way: "Memprof stats are updated in a deferred way. This is required in order to present a consistent picture in the situation when allocs are coming continuously and frees are coming in batches afterwards. Several consecutive GCs push the update pipeline forward. That's what you observe. So if you profile a live server then any sample will give you a consistent snapshot. However, if the program completes some activity and you want to collect the snapshot after this activity, then you need to execute 2 or 3 GCs before the collection."

Blocking Profiler

The blocking profiler shows where goroutine block waiting on synchronization primitives (including timer channels). You can collect it in similar ways as CPU profile: with `go test --blockprofile` (https://pkg.go.dev/cmd/go/#hdr-Description_of_testing_flags), with net/http/pprof via http://myserver:6060:/debug/pprof/block or by calling runtime/pprof.Lookup("block").WriteTo.

But there is significant caveat -- the blocking profiler is not enabled by default. `go test --blockprofile` will enable it for you automatically. However, if you use `net/http/pprof` or `runtime/pprof`, you need to enable it manually (otherwise the profile will be empty). To enable the blocking profiler call runtime.SetBlockProfileRate. SetBlockProfileRate controls the fraction of goroutine blocking events that are reported in the blocking profile. The profiler aims to sample an average of one blocking event per the specified amount of nanoseconds spent blocked. To include every blocking event in the profile, set the rate to 1.

If a function contains several blocking operations and it's not obvious which one leads to blocking, use `--lines` flag to pprof.

Note that not all blocking is bad. When a goroutine blocks, the underlying worker thread simply switches to another goroutine. So blocking in the cooperative Go environment is very different from blocking on a mutex in a non-cooperative systems (e.g. typical C++ or Java threading libraries, where blocking leads to thread idling and expensive thread context switches). To give you some feeling, let's consider some examples.

Blocking on a time.Ticker is usually OK. If a goroutine blocks on a Ticker for 10 seconds, you will see 10 seconds of blocking in the profile, which is perfectly fine. Blocking on `sync.WaitGroup` is frequently OK. For example, is a task takes 10 seconds, the goroutine waiting on a WaitGroup for completion will account for 10 seconds of blocking in the profile. Blocking on sync.Cond may or may not be OK, depending on the situation. Consumer blocking on a channel suggests slow producers or lack of work. Producer blocking on a channel suggests that consumers are slower, but this is frequently OK. Blocking on a channel-based semaphore shows how much goroutines are gated on the semaphore. Blocking on a sync.Mutex or sync.RWMutex is usually bad. You can use `--ignore` flag to pprof to exclude known uninteresting blocking from a profile during visualization.

Blocking of goroutines can lead to two negative consequences:

1. Program does not scale with processors due to lack of work. Scheduler Trace can help to identify this case.
2. Excessive goroutine blocking/unblocking consumes CPU time. CPU Profiler can help to identify this case (look at the System component).

Here are some common suggestions that can help to reduce goroutine blocking:

1. Use sufficiently buffered channels in producer-consumer scenarios. Unbuffered channels substantially limit available parallelism in the program.
2. Use `sync.RWMutex` instead of `sync.Mutex` for read-mostly workloads. Readers never block other readers in `sync.RWMutex`, even on implementation level.
3. In some cases it's possible to remove mutexes entirely by using copy-on-write technique. If the protected data structure is modified infrequently and it's feasible to make copies of it, then it can be updated as follows:

   ```
   type Config struct {
       Routes   map[string]net.Addr
       Backends []net.Addr
   }

   var config atomic.Value  // actual type is *Config

   // Worker goroutines use this function to obtain the current config.
     // UpdateConfig must be called at least once before this func.
   func CurrentConfig() *Config {
       return config.Load().(*Config)
   }

   // Background goroutine periodically creates a new Config object
   // as sets it as current using this function.
   func UpdateConfig(cfg *Config) {
     config.Store(cfg)
   }
   ```

   This pattern prevents the writer from blocking readers during update.
4. Partitioning is another general technique for reducing contention/blocking on shared mutable data structures. Below is an example of how to partition a hashmap:

```
type Partition struct {
    sync.RWMutex
    m map[string]string
}

const partCount = 64
var m [partCount]Partition

func Find(k string) string {
    idx := hash(k) % partCount
    part := &m[idx]
    part.RLock()
    v := part.m[k]
    part.RUnlock()
    return v
}
```

5. Local caching and batching of updates can help to reduce contention on un-partitionable data structures. Below you can see how to batch sends to a channel:

```
const CacheSize = 16

type Cache struct {
    buf [CacheSize]int
    pos int
}

func Send(c chan [CacheSize]int, cache *Cache, value int) {
    cache.buf[cache.pos] = value
    cache.pos++
    if cache.pos == CacheSize {
        c <- cache.buf
        cache.pos = 0
    }
}
```

   This technique is not limited to channels. It can be used to batch updates to a map, batch allocations, etc.

6. Use sync.Pool for freelists instead of chan-based or mutex-protected freelists. sync.Pool uses smart techniques internally to reduce blocking.

Goroutine Profiler

The goroutine profiler simply gives you current stacks of all live goroutines in the process. It can be handy to debug load balancing issues (see Scheduler Trace section below), or to debug deadlocks. The profile makes sense only for a running app, so go test does not expose it. You can collect the profile with net/http/pprof via http://myserver:6060:/debug/pprof/goroutine, and visualize it to svg/pdf or by calling runtime/pprof.Lookup("goroutine").WriteTo. But the most useful way is to type http://myserver:6060:/debug/pprof/goroutine?debug=2 in your browser, which will give you symbolized stacks similar to what you see when a program crashes. Note that goroutines in "syscall" state consume an OS thread, other goroutines do not (except for goroutines that called runtime.LockOSThread, which is, unfortunately, not visible in the profile). Note that goroutines in "IO wait" state also do not consume threads, they are parked on non-blocking network poller (which uses epoll/kqueue/GetQueuedCompletionStatus to unpark goroutines later).

Garbage Collector Trace

Aside from the profiling tools, there is another kind of tools available -- tracers. They allow to trace garbage collections, memory allocator and goroutine scheduler state. To enable the garbage collector (GC) trace, run the program with `GODEBUG=gctrace=1` environment variable:

```
$ GODEBUG=gctrace=1 ./myserver
```

Then the program will print output similar to the following during execution:

```
gc9(2): 12+1+744+8 us, 2 -> 10 MB, 108615 (593983-485368) objects, 4825/3620/0 sweeps,
0(0) handoff, 6(91) steal, 16/1/0 yields
gc10(2): 12+6769+767+3 us, 1 -> 1 MB, 4222 (593983-589761) objects, 4825/0/1898
sweeps, 0(0) handoff, 6(93) steal, 16/10/2 yields
gc11(2): 799+3+2050+3 us, 1 -> 69 MB, 831819 (1484009-652190) objects, 4825/691/0
sweeps, 0(0) handoff, 5(105) steal, 16/1/0 yields
```

Let's consider the meaning of these numbers. One line per GC is printed. The first number ("gc9") is the number of GC (this is the 9-th GC since program start). The number in parens ("(2)") is the number of worker threads participated in the GC. The next 4 numbers ("12+1+744+8 us") mean stop-the-world, sweeping, marking and waiting for worker threads to finish, in microseconds, respectively. The next 2 numbers ("2 -> 10 MB") mean size of live heap after the previous GC and full heap size (including garbage) before the current GC. The next 3 numbers ("108615 (593983-485368) objects") are total number of objects in heap (including garbage) and total number of memory allocation and free operations. The next 3 numbers ("4825/3620/0 sweeps") characterize sweep phase (of the previous GC): there were total 4825 memory spans, 3620 were swept on demand or in background, 0 were swept during stop-the-world phase (the rest were unused spans). The next 4 numbers ("0(0) handoff, 6(91) steal") characterize load balancing during parallel mark phase: there were 0 object handoff operations (0 objects were handoff), and 6 steal operations (91 objects were stolen). The last 3 numbers ("16/1/0 yields") characterize effectiveness of parallel mark phase: there were total of 17 yield operations during waiting for another thread.

The GC is mark-and-sweep type. Total GC can be expressed as:

```
Tgc = Tseq + Tmark + Tsweep
```

where Tseq is time to stop user goroutines and some preparation activities (usually small); Tmark is heap marking time, marking happens when all user goroutines are stopped, and thus can significantly affect latency of processing; Tsweep is heap sweeping time, sweeping generally happens concurrently with normal program execution, and so is not so critical for latency.

Marking time can be approximately expressed as:

```
Tmark = C1*Nlive + C2*MEMlive_ptr + C3*Nlive_ptr
```

where Nlive is the number of live objects in the heap during GC, `MEMlive_ptr` is the amount of memory occupied by live objects with pointers, `Nlive_ptr` is the number of pointers in live objects.

Sweeping time can be approximately expressed as:

```
Tsweep = C4*MEMtotal + C5*MEMgarbage
```

where `MEMtotal` is the total amount of heap memory, `MEMgarbage` is the amount of garbage in the heap.

Next GC happens after the program has allocated an extra amount of memory proportional to the amount already in use. The proportion is controlled by GOGC environment variable (100 by default). If GOGC=100 and the program is using 4M of heap memory, runtime will trigger GC again when the program gets to 8M. This keeps the GC cost in linear proportion to the allocation cost. Adjusting GOGC changes the linear constant and also the amount of extra memory used.

Only sweeping depends on total size of the heap, and sweeping happens concurrently with normal program execution. So it can make sense to set GOGC to a higher value (200, 300, 500, etc) if you can afford extra memory consumption. For example, GOGC=300 can reduce garbage collection overhead by up to 2 times while keeping latencies the same (at the cost of 2 times larger heap).

GC is parallel and generally scales well with hardware parallelism. So it can make sense to set GOMAXPROCS to higher value even for sequential programs just to speed up garbage collections. However, note that number of garbage collector threads is currently bounded by 8.

Memory Allocator Trace

Memory allocator traces simply dumps all memory allocation and free operations onto console. It's enabled with GODEBUG=allocfreetrace=1 environment variable. The output looks along the lines of:

```
tracealloc(0xc208062500, 0x100, array of parse.Node)
goroutine 16 [running]:
runtime.mallocgc(0x100, 0x3eb7c1, 0x0)
    runtime/malloc.goc:190 +0x145 fp=0xc2080b39f8
runtime.growslice(0x31f840, 0xc208060700, 0x8, 0x8, 0x1, 0x0, 0x0, 0x0)
    runtime/slice.goc:76 +0xbb fp=0xc2080b3a90
text/template/parse.(*Tree).parse(0xc2080820e0, 0xc208023620, 0x0, 0x0)
    text/template/parse/parse.go:289 +0x549 fp=0xc2080b3c50
...

tracefree(0xc208002d80, 0x120)
goroutine 16 [running]:
runtime.MSpan_Sweep(0x73b080)
        runtime/mgc0.c:1880 +0x514 fp=0xc20804b8f0
runtime.MCentral_CacheSpan(0x69c858)
        runtime/mcentral.c:48 +0x2b5 fp=0xc20804b920
runtime.MCache_Refill(0x737000, 0xc200000012)
        runtime/mcache.c:78 +0x119 fp=0xc20804b950
...
```

The trace contains address of the memory block, size, type, goroutine id and the stack trace. It's probably more useful for debugging, but can give very fine-grained info for allocation optimizations as well.

Scheduler Trace

Scheduler trace can provide insights into dynamic behavior of the goroutine scheduler and allow to debug load balancing and scalability issues. To enable the scheduler trace trace, run the program with GODEBUG=schedtrace=1000 environment variable (the value means period of output, in ms, in this case it's once per second):

```
$ GODEBUG=schedtrace=1000 ./myserver
```

Then the program will print output similar to the following during execution:

```
SCHED 1004ms: gomaxprocs=4 idleprocs=0 threads=11 idlethreads=4 runqueue=8 [0 1 0 3]
SCHED 2005ms: gomaxprocs=4 idleprocs=0 threads=11 idlethreads=5 runqueue=6 [1 5 4 0]
SCHED 3008ms: gomaxprocs=4 idleprocs=0 threads=11 idlethreads=4 runqueue=10 [2 2 2 1]
```

The first number ("1004ms") is time since program start. Gomaxprocs is the current value of GOMAXPROCS. Idleprocs is the number of idling processors (the rest are executing Go code). Threads is the total number of worker threads created by the scheduler (threads can be in 3 states: execute Go code (gomaxprocs-idleprocs), execute syscalls/cgocalls or idle). Idlethreads is the number of idling worker threads. Runqueue is the length of global queue with runnable goroutines. The numbers in square brackets ("[0 1 0 3]") are lengths of per-processor queues with runnable goroutines. Sum of lengths of global and local queues represents the total number of goroutines available for execution.

Note: You can combine any of the tracers as GODEBUG=gctrace=1,allocfreetrace=1,schedtrace=1000.

Note: There is also detailed scheduler trace, which you can enable with GODEBUG=schedtrace=1000,scheddetail=1. It prints detailed info about every goroutine, worker thread and processor. We won't describe its format here as it's mainly useful for scheduler developers; but you can find details in src/pkg/runtime/proc.c.

The scheduler trace is useful when a program does not scale linearly with GOMAXPROCS and/or does not consume 100% of CPU time. The ideal situation is when all processors are busy executing Go code, number of threads is reasonable, there is plenty of work in all queues and the work is reasonably evenly distributed:

```
gomaxprocs=8 idleprocs=0 threads=40 idlethreads=5 runqueue=10 [20 20 20 20 20 20 20
20]
```

A bad situation is when something of the above does not hold. For example the following sample demonstrates shortage of work to keep all processors busy:

```
gomaxprocs=8 idleprocs=6 threads=40 idlethreads=30 runqueue=0 [0 2 0 0 0 1 0 0]
```

Note: use OS-provided means to measure actual CPU utilization as the ultimate characteristic. On Unix family of operating system it is top command; on Windows it is Task Manager.

You can use the goroutine profiler to understand where goroutines block in the case of work shortage. Note that load imbalance is not ultimately bad as long as all processors are busy, it will just cause some moderate load balancing overheads.

Memory Statistics

Go runtime exposes coarse-grained memory statistics via runtime.ReadMemStats function. The statistics are also exposed via net/http/pprof at the bottom of http://myserver:6060/debug/pprof/heap?debug=1. The statistics are described here. Some of the interesting fields are:

1. HeapAlloc - current heap size.
2. HeapSys - total heap size.
3. HeapObjects - total number of objects in the heap.
4. HeapReleased - amount of memory released to the OS; runtime releases to the OS memory unused for 5 minutes, you can force this process with runtime/debug.FreeOSMemory.
5. Sys - total amount of memory allocated from OS.

6. Sys-HeapReleased - effective memory consumption of the program.
7. StackSys - memory consumed for goroutine stacks (note that some stacks are allocated from heap and are not accounted here, unfortunately there is no way to get total size of stacks (https://code.google.com/p/go/issues/detail?id=7468)).
8. MSpanSys/MCacheSys/BuckHashSys/GCSys/OtherSys - amount of memory allocated by runtime for various auxiliary purposes; they are generally not interesting, unless they are too high.
9. PauseNs - durations of last garbage collections.

Heap Dumper

The last available tool is heap dumper, it can write state of the whole heap into a file for future exploration. It can be useful for identifying memory leaks and getting insights into program memory consumption.

First, you need to write the dump using runtime/debug.WriteHeapDump function:

```
	f, err := os.Create("heapdump")
	if err != nil { ... }
	debug.WriteHeapDump(f.Fd())
```

Then you can either render it to a dot file with graphical representation of the heap or convert it to hprof format. To render it to a dot file:

```
$ go get github.com/randall77/hprof/dumptodot
$ dumptodot heapdump mybinary > heap.dot
```

and open `heap.dot` with Graphviz.

To convert it to `hprof` format:

```
$ go get github.com/randall77/hprof/dumptohprof
$ dumptohprof heapdump heap.hprof
$ jhat heap.hprof
```

and navigate your browser to http://myserver:7000.

Concluding Remarks

Optimization is an open problem, there are simple recipes that you can use to improve performance. Sometimes optimization requires complete re-architecture of the program. But we hope that the tools will be a valuable addition to your toolbox, that you can use to at least analyze and understand what happens. Profiling Go Programs is a good tutorial on usage of CPU and memory profilers to optimize a simple program.