Scalable Go Scheduler Design Doc

Dmitry Vyukov
dvyukov@google.com
May 2, 2012

The document assumes some prior knowledge of the Go language and current goroutine scheduler implementation.

Problems with current scheduler

Current goroutine scheduler limits scalability of concurrent programs written in Go, in particular, high-throughput servers and parallel computational programs. Vtocc server maxes out at 70% CPU on 8-core box, while profile shows 14% is spent in runtime.futex(). In general, the scheduler may inhibit users from using idiomatic fine-grained concurrency where performance is critical.

What's wrong with current implementation:

1. Single global mutex (Sched.Lock) and centralized state. The mutex protects all goroutine-related operations (creation, completion, rescheduling, etc).
2. Goroutine (G) hand-off (G.nextg). Worker threads (M's) frequently hand-off runnable goroutines between each other, this may lead to increased latencies and additional overheads. Every M must be able to execute any runnable G, in particular the M that just created the G.
3. Per-M memory cache (M.mcache). Memory cache and other caches (stack alloc) are associated with all M's, while they need to be associated only with M's running Go code (an M blocked inside of syscall does not need mcache). A ratio between M's running Go code and all M's can be as high as 1:100. This leads to excessive resource consumption (each MCache can suck up up to 2M) and poor data locality.
4. Aggressive thread blocking/unblocking. In presence of syscalls worker threads are frequently blocked and unblocked. This adds a lot of overhead.

Design

Processors

The general idea is to introduce a notion of P (Processors) into runtime and implement

work-stealing scheduler on top of Processors.
M represents OS thread (as it is now). P represents a resource that is required to execute Go code. When M executes Go code, it has an associated P. When M is idle or in syscall, it does need P.
There is exactly GOMAXPROCS P's. All P's are organized into an array, that is a requirement of work-stealing. GOMAXPROCS change involves stop/start the world to resize array of P's.
Some variables from sched are de-centralized and moved to P. Some variables from M are moved to P (the ones that relate to active execution of Go code).

```
struct P
{
      Lock;
      G *gfree; // freelist, moved from sched
      G *ghead; // runnable, moved from sched
      G *gtail;
      MCache *mcache; // moved from M
      FixAlloc *stackalloc; // moved from M
      uint64 ncgocall;
      GCStats gcstats;
      // etc
      ...
};

P *allp; // [GOMAXPROCS]
```

There is also a lock-free list of idle P's:

```
P *idlep; // lock-free list
```

When an M is willing to start executing Go code, it must pop a P form the list. When an M ends executing Go code, it pushes the P to the list. So, when M executes Go code, it necessary has an associated P. This mechanism replaces sched.atomic (mcpu/mcpumax).

Scheduling

When a new G is created or an existing G becomes runnable, it is pushed onto a list of runnable goroutines of current P. When P finishes executing G, it first tries to pop a G from own list of runnable goroutines; if the list is empty, P chooses a random victim (another P) and tries to steal a half of runnable goroutines from it.

Syscalls/M Parking and Unparking

When an M creates a new G, it must ensure that there is another M to execute the G (if not all

M’s are already busy). Similarly, when an M enters syscall, it must ensure that there is another M to execute Go code.
There are two options, we can either promptly block and unblock M’s, or employ some spinning. Here is inherent conflict between performance and burning unnecessary CPU cycles. The idea is to use spinning and do burn CPU cycles. However, it should not affect programs running with GOMAXPROCS=1 (command line utilities, appengine, etc).
Spinning is two-level: (1) an idle M with an associated P spins looking for new G’s, (2) an M w/o an associated P spins waiting for available P’s. There are at most GOMAXPROCS spinning M’s (both (1) and (2)). Idle M’s of type (1) do not block while there are idle M’s of type (2).
When a new G is spawned, or M enters syscall, or M transitions from idle to busy, it ensures that there is at least 1 spinning M (or all P’s are busy). This ensures that there are no runnable G’s that can be otherwise running; and avoids excessive M blocking/unblocking at the same time.
Spinning is mostly passive (yield to OS, sched_yield()), but may include a little bit of active spinning (loop burnging CPU) (requires investigation and tuning).

Termination/Deadlock Detection

Termination/deadlock detection is more problematic in a distributed system. The general idea is to do the checks only when all P’s are idle (global atomic counter of idle P’s), this allows to do more expensive checks that involve aggregation of per-P state.
No details yet.

LockOSThread

This functionality is not performance-critical.
1. Locked G become non-runnable (Gwaiting). M instantly returns P to idle list, wakes up another M and blocks.
2. Locked G becomes runnable (and reaches head of the runq). Current M hands off own P and locked G to the M associated with the locked G, and unblocks it. Current M becomes idle.

Idle G

This functionality is not performance-critical.
There is a global queue of (or a single?) idle G. An M that looks for work checks the queue after several unsuccessful steal attempts.

Implementation Plan

The goal is to split the whole thing into minimal parts that can be independently reviewed and submitted.
1. Introduce the P struct (empty for now); implement allp/idlep containers (idlep is mutex-protected for starters); associate a P with M running Go code. Global mutex and atomic

state is still preserved.
2. Move G freelist to P.
3. Move mcache to P.
4. Move stackalloc to P.
5. Move ncgocall/gcstats to P.
6. Decentralize run queue, implement work-stealing. Eliminate G hand off. Still under global mutex.
7. Remove global mutex, implement distributed termination detection, LockOSThread.
8. Implement spinning instead of prompt blocking/unblocking.
The plan may turn out to not work, there are a lot of unexplored details.

Potential Further Improvements

1. Try out LIFO scheduling, this will improve locality. However, it still must provide some degree of fairness and gracefully handle yielding goroutines.
2. Do not allocate G and stack until the goroutine first runs. For a newly created goroutine we need just callerpc, fn, narg, nret and args, that is, about 6 words. This will allow to create a lot of running-to-completion goroutines with significantly lower memory overhead.
4. Better locality of G-to-P. Try to enqueue an unblocked G to a P on which it was last running.
5. Better locality of P-to-M. Try to execute P on the same M it was last running.
6. Throttling of M creation. The scheduler can be easily forced to create thousands of M's per second until OS refuses to create more threads. M’s must be created promptly up to k*GOMAXPROCS, after that new M’s may added by a timer.

Random Notes

- GOMAXPROCS won’t go away as a result of this work.