CR1

// go test -run none -bench . -benchtime 3s -benchmem

// Basic benchmark test.

package basic

import (

"fmt"

"testing"

)

var gs string

// BenchmarkSprint tests the performance of using Sprint.

func BenchmarkSprint(b \*testing.B) {

var s string

for i := 0; i < b.N; i++ {

s = fmt.Sprint("hello")

}

gs = s

}

// BenchmarkSprint tests the performance of using Sprintf.

func BenchmarkSprintf(b \*testing.B) {

var s string

for i := 0; i < b.N; i++ {

s = fmt.Sprintf("hello")

}

gs = s

}

CR2

// go test -run none -bench . -benchtime 3s -benchmem

// go test -run none -bench BenchmarkSprint/none -benchtime 3s -benchmem

// go test -run none -bench BenchmarkSprint/format -benchtime 3s -benchmem

// Basic sub-benchmark test.

package basic

import (

"fmt"

"testing"

)

var gs string

// BenchmarkSprint tests all the Sprint related benchmarks as

// sub benchmarks.

func BenchmarkSprint(b \*testing.B) {

b.Run("none", benchSprint)

b.Run("format", benchSprintf)

}

// benchSprint tests the performance of using Sprint.

func benchSprint(b \*testing.B) {

var s string

for i := 0; i < b.N; i++ {

s = fmt.Sprint("hello")

}

gs = s

}

// benchSprintf tests the performance of using Sprintf.

func benchSprintf(b \*testing.B) {

var s string

for i := 0; i < b.N; i++ {

s = fmt.Sprintf("hello")

}

gs = s

}

CR3

// Sample program to show you need to validate your benchmark results.

package main

import (

"math"

"runtime"

"sync"

"testing"

)

// n contains the data to sort.

var n []int

// Generate the numbers to sort.

func init() {

for i := 0; i < 1000000; i++ {

n = append(n, i)

}

}

func BenchmarkSingle(b \*testing.B) {

for i := 0; i < b.N; i++ {

single(n)

}

}

func BenchmarkUnlimited(b \*testing.B) {

for i := 0; i < b.N; i++ {

unlimited(n)

}

}

func BenchmarkNumCPU(b \*testing.B) {

for i := 0; i < b.N; i++ {

numCPU(n, 0)

}

}

// single uses a single goroutine to perform the merge sort.

func single(n []int) []int {

// Once we have a list of one we can begin to merge values.

if len(n) <= 1 {

return n

}

// Split the list in half.

i := len(n) / 2

// Sort the left side.

l := single(n[:i])

// Sort the right side.

r := single(n[i:])

// Place things in order and merge ordered lists.

return merge(l, r)

}

// unlimited uses a goroutine for every split to perform the merge sort.

func unlimited(n []int) []int {

// Once we have a list of one we can begin to merge values.

if len(n) <= 1 {

return n

}

// Split the list in half.

i := len(n) / 2

// Maintain the ordered left and right side lists.

var l, r []int

// For each split we will have 2 goroutines.

var wg sync.WaitGroup

wg.Add(2)

// Sort the left side concurrently.

go func() {

l = unlimited(n[:i])

wg.Done()

}()

// Sort the right side concurrenyly.

go func() {

r = unlimited(n[i:])

wg.Done()

}()

// Wait for the spliting to end.

wg.Wait()

// Place things in order and merge ordered lists.

return merge(l, r)

}

// numCPU uses the same number of goroutines that we have cores

// to perform the merge sort.

func numCPU(n []int, lvl int) []int {

// Once we have a list of one we can begin to merge values.

if len(n) <= 1 {

return n

}

// Split the list in half.

i := len(n) / 2

// Maintain the ordered left and right side lists.

var l, r []int

// Cacluate how many levels deep we can create goroutines.

// On an 8 core machine we can keep creating goroutines until level 4.

// Lvl 0 1 Lists 1 Goroutine

// Lvl 1 2 Lists 2 Goroutines

// Lvl 2 4 Lists 4 Goroutines

// Lvl 3 8 Lists 8 Goroutines

// Lvl 4 16 Lists 16 Goroutines

// On 8 core machine this will produce the value of 3.

maxLevel := int(math.Log2(float64(runtime.NumCPU())))

// We don't need more goroutines then we have logical processors.

if lvl <= maxLevel {

lvl++

// For each split we will have 2 goroutines.

var wg sync.WaitGroup

wg.Add(2)

// Sort the left side concurrently.

go func() {

l = numCPU(n[:i], lvl)

wg.Done()

}()

// Sort the right side concurrenyly.

go func() {

r = numCPU(n[i:], lvl)

wg.Done()

}()

// Wait for the spliting to end.

wg.Wait()

// Place things in order and merge ordered lists.

return merge(l, r)

}

// Sort the left and right side on this goroutine.

l = numCPU(n[:i], lvl)

r = numCPU(n[i:], lvl)

// Place things in order and merge ordered lists.

return merge(l, r)

}

// merge performs the merging to the two lists in proper order.

func merge(l, r []int) []int {

// Declare the sorted return list with the proper capacity.

ret := make([]int, 0, len(l)+len(r))

// Compare the number of items required.

for {

switch {

case len(l) == 0:

// We appended everything in the left list so now append

// everything contained in the right and return.

return append(ret, r...)

case len(r) == 0:

// We appended everything in the right list so now append

// everything contained in the left and return.

return append(ret, l...)

case l[0] <= r[0]:

// First value in the left list is smaller than the

// first value in the right so append the left value.

ret = append(ret, l[0])

// Slice that first value away.

l = l[1:]

default:

// First value in the right list is smaller than the

// first value in the left so append the right value.

ret = append(ret, r[0])

// Slice that first value away.

r = r[1:]

}

}

}

CR4

// go test -run none -bench . -benchtime 3s -benchmem

// Package prediction provides code to show how branch

// prediction can affect performance.

package prediction

import (

"math/rand"

"testing"

)

// crunch is used to perform branch instructions.

func crunch(data []uint8) uint8 {

var sum uint8

for \_, v := range data {

if v < 128 {

sum--

} else {

sum++

}

}

return sum

}

var fa uint8

// BenchmarkPredictable runs the test when the branch is predictable.

func BenchmarkPredictable(b \*testing.B) {

data := make([]uint8, 1024)

b.ResetTimer()

var a uint8

for i := 0; i < b.N; i++ {

a = crunch(data)

}

fa = a

}

// BenchmarkUnpredictable runs the test when the branch is random.

func BenchmarkUnpredictable(b \*testing.B) {

data := make([]uint8, 1024)

rand.Seed(0)

// Fill data with (pseudo) random noise

for i := range data {

data[i] = uint8(rand.Uint32())

}

b.ResetTimer()

var a uint8

for i := 0; i < b.N; i++ {

a = crunch(data)

}

fa = a

}

CR5

// Package caching provides code to show why Data Oriented Design matters. How

// data layouts matter more to performance than algorithm efficiency.

package caching

import "fmt"

// Create a square matrix of 16,777,216 bytes.

const (

rows = 4 \* 1024

cols = 4 \* 1024

)

// matrix represents a matrix with a large number of

// columns per row.

var matrix [rows][cols]byte

// data represents a data node for our linked list.

type data struct {

v byte

p \*data

}

// list points to the head of the list.

var list \*data

func init() {

var last \*data

// Create a link list with the same number of elements.

for row := 0; row < rows; row++ {

for col := 0; col < cols; col++ {

// Create a new node and link it in.

var d data

if list == nil {

list = &d

}

if last != nil {

last.p = &d

}

last = &d

// Add a value to all even elements.

if row%2 == 0 {

matrix[row][col] = 0xFF

d.v = 0xFF

}

}

}

// Count the number of elements in the link list.

var ctr int

d := list

for d != nil {

ctr++

d = d.p

}

fmt.Println("Elements in the link list", ctr)

fmt.Println("Elements in the matrix", rows\*cols)

}

// LinkedListTraverse traverses the linked list linearly.

func LinkedListTraverse() int {

var ctr int

d := list

for d != nil {

if d.v == 0xFF {

ctr++

}

d = d.p

}

return ctr

}

// ColumnTraverse traverses the matrix linearly down each column.

func ColumnTraverse() int {

var ctr int

for col := 0; col < cols; col++ {

for row := 0; row < rows; row++ {

if matrix[row][col] == 0xFF {

ctr++

}

}

}

return ctr

}

// RowTraverse traverses the matrix linearly down each row.

func RowTraverse() int {

var ctr int

for row := 0; row < rows; row++ {

for col := 0; col < cols; col++ {

if matrix[row][col] == 0xFF {

ctr++

}

}

}

return ctr

}

// go test -run none -bench . -benchtime 3s

// Tests to show how Data Oriented Design matters.

package caching

import "testing"

var fa int

// Capture the time it takes to perform a link list traversal.

func BenchmarkLinkListTraverse(b \*testing.B) {

var a int

for i := 0; i < b.N; i++ {

a = LinkedListTraverse()

}

fa = a

}

// Capture the time it takes to perform a column traversal.

func BenchmarkColumnTraverse(b \*testing.B) {

var a int

for i := 0; i < b.N; i++ {

a = ColumnTraverse()

}

fa = a

}

// Capture the time it takes to perform a row traversal.

func BenchmarkRowTraverse(b \*testing.B) {

var a int

for i := 0; i < b.N; i++ {

a = RowTraverse()

}

fa = a

}

CR6

// Tests to show the effect of false sharing on concurrent

// memory writes.

package falseshare

import (

"sync"

"testing"

)

// cnt represents a data value for our counter.

type cnt struct {

counter int64

}

// Create an array of 8 counters.

var countersPad [8]cnt

// BenchmarkGlobal tests the performance of 8 goroutines

// incrementing the global counter in parallel.

func BenchmarkGlobal(b \*testing.B) {

// Create and set the WaitGroup.

var wg sync.WaitGroup

for i := 0; i < b.N; i++ {

// Add the 8 goroutines we will create.

wg.Add(8)

// Create the 8 goroutines.

for g := 0; g < 8; g++ {

// Have each goroutine loop and increment

// their specific global counter.

go func(i int) {

for {

// Increment the specific global counter.

countersPad[i].counter++

// Check if we have incremented it enough.

if countersPad[i].counter%1e6 == 0 {

// Report we are done and terminate.

wg.Done()

return

}

}

}(g)

}

// Wait for all the goroutines to finish.

wg.Wait()

}

}

// BenchmarkLocal tests the performance of 8 goroutines

// incrementing their local counter.

func BenchmarkLocal(b \*testing.B) {

// Create and set the WaitGroup.

var wg sync.WaitGroup

for i := 0; i < b.N; i++ {

// Add the 8 goroutines we will create.

wg.Add(8)

// Create the 8 goroutines.

for g := 0; g < 8; g++ {

// Have each goroutine loop and increment

// their specific local counter.

go func(i int) {

// Init the local counter.

var counter int64

for {

// Increment the local counter.

counter++

// Check if we have incremented it enough.

if counter%1e6 == 0 {

// Write the final counter to the

// specific global counter.

countersPad[i].counter = counter

// Report we are done and terminate.

wg.Done()

return

}

}

}(g)

}

// Wait for all the goroutines to finish.

wg.Wait()

}

}