Parallel Computing in Go

Jordon Phillips

May 27, 2014

The Environment

Go is a language built with concurrency in mind. Instead of having threads, processes, coroutines, etc. it has what it calls goroutines. On a basic level, a goroutine is an asynchronously executing function. They are very small, requiring only a small amount of stack space to be initialized. One of the really nice things about goroutines is that the Go runtime multiplexes all of them onto multiple OS threads, meaning that if a goroutine blocks others can continue to run. All the goroutines for a given program share the same address space, but for most applications they should not share variables.

In Go, the idiomatic way of sharing data is by communicating over channels. Channels can be thought of like fixed length queues. A goroutine can put data into the channel or it can pull data out. One thing to consider that differs greatly from something like MPI is that you have no guarantee of who gets what data. If you want to limit a certain kind of data to a certain set of goroutines, then you have create a channel just for them and pass the data on that channel. Once you're done with that channel, or any channel, you can close it and any goroutine still waiting on it will recieve a message that the channel has closed.

Despite that, it is still possible to share data directly. An anonymous function retains access to the defining function's variables, and so they can be used to share data directly. This is not encouraged in many cases, since it it adds the risk of data races. The ability is there for those who need it, and Go provides several other ways of synchronization besides channels, such as locks, that help out in such situations. It is important to know that it lacks one thing that is commonly used in communication: identifiers. Goroutines do not have any identifiers unless you manually supply some. Trying to use a process id would fail, since goroutines may jump processes if their previous one was too busy.

The Language

An introduction to the language, or tool, or library, or compiler directives, you are using. This should give an explanation of the fundamental ideas of your tool and should give the reader enough information that he could implement some basic algorithms.

Go, like C, is an imperitive language. In many respects it looks and works like the many other languages in that class, so a person who is familiar with C or C++ would have a relatively simple process of adjustment. Unlike either of those languages, it's garbage collected, so you don't have to worry about freeing memory. If you would like to try it out without bothering to install it, you can play around a bit on the Go Playground. Examples presented here will include links to snippets there.

A much, much better primer on the language can be found here

Hello World

```
1  package main
2
3  import (
4    "fmt"
5  )
6
7  func main() {
8    fmt.Println("Hello, playground")
9  }
```

Listing 1: Hello World

Starting from the top, we have our package declaration. For an executable package, the name has to be main. All files with the same package name share code directly without import. The import keyword is used to input a list of packages which are newline separated. Functions are declared with 'func'. Functions whose name begins with a capital letter can be accessed by any importers, functions whose name begins with a lowercase letter are private to the package.

Variables and Slices

```
1  package main
2
3  import "fmt"
4
5  func main() {
6    slice := []float64{4, 6, 3, 2}
7    sum := Sum(slice)
8    fmt.Printf("The sum is: %.2f\n", sum)
9
10   slice2 := make([]float64, 4)
11   sum = Sum(slice2)
12   fmt.Printf("The sum is: %.2f\n", sum)
13
14   var sum3 float64
15   sum3 = Sum(slice)
16   fmt.Printf("The sum is: %.2f\n", sum3)
17 }
18
19 func Sum(slice []float64) (sum float64) {
20   for _, val := range slice {
```

```
21      sum += val
22    }
23    return
24  }
```

Listing 2: Variables and Slices

To get some basic computation done, all you really need are some variables and some slices. In Go, slices are the default datatype used for lists of elements. There are a lot of differences between a slice and an array, but for the sake of brevity we'll assume they're equivalent.

A slice can be allocated in two ways, with the 'make' function or with an inline initializtion. While it can be accessed in the same manner as an array, it can also be accessed from a for loop with the 'range' function.

Other than that, slices are like any other variable. To instantiate a variable, you use ':=' and to reassign you use '='. If you don't want to provide an initial value, you can instantiate in the same way as is done on line 14. You'll notice that on line 14 and in the function declaration on line 19, the type follows the name rather than the other way around. This is done to improve the readablitiy of the source code. This may seem silly, but the Go authors have their reasons.

Go functions can pre-define the names of their return values. If done so, the variables will be initialzed to their zero values and will be available for access within the function. A return will still be needed, but arguments become optional if this is done.

On line 19 you can see an underscore. The underscore by itself is a reserved word that represents a return value you don't care about. This is important because functions in Go can have multiple return values and unused variables are compiler **errors**. (In fact, there are no compiler warnings. There are only errrors.)

Parallelizing

Now it's time to start making things concurrent! This is crazy easy in Go and very simple to understand. So easy that I'll let the comments speak for themselves!

```
1   package main
2
3   import "fmt"
4
5   func main() {
6     // Running a single function in parallel.
7     // A non-buffered channel is basically like a semaphore
8     done := make(chan bool)
9     slice := []float64{4, 6, 3, 2}
10    go SumSemaphore(slice, done)
11
12    // We could assign the value to a local variable,
13    // but in this case it is irrelevant
14    <-done
15
```

```
16    // Now we make a buffered channel to hold 3 pieces of data
17    sumChannel := make(chan []float64, 3)
18    go SumConsumer(sumChannel, done)
19
20    // We feed our input data into the channel
21    sumChannel <- slice
22    sumChannel <- []float64{1, 2, 3, 4, 5, 6}
23
24    // And close it when we're done sending input
25    close(sumChannel)
26    <-done
27
28    // We can also have inline anonymous functions
29    go func(slice2 []float64) {
30      Sum(slice) // The function is within the parent's scope
31      Sum(slice2)
32      done <- true
33    } ([]float64{4,4,4,4}) // arguments are given at the end
34    <-done
35  }
36
37  // If you have more than one consumer, you are NOT guaranteed which
38  // consumer will pop any given piece of data from the channel.
39  func SumConsumer(data chan []float64, done chan bool) {
40    // The range function will automatically terminate
41    // when the channel is closed.
42    for slice := range data {
43      Sum(slice)
44    }
45    done <- true
46  }
47
48  func SumSemaphore(slice []float64, done chan bool) {
49    Sum(slice)
50    // Push to the channel to singal completion
51    done <- true
52  }
53
54  func Sum(slice []float64) {
55    var sum float64 = 0
56    for _, val := range slice {
57      sum += val
58    }
59    fmt.Printf("The sum is: %.2f\n", sum)
60  }
```

Listing 3: Concurrent Go

There is one gotcha: by default, Go is *concurrent*, not parallel. That is, it only uses one kernel level thread. There are plans to improve the Go runtime so that it will be parallel by default, but for now you need to manually specify the number of cores to use, like so:

```
1  package main
2
3  import (
4    "fmt"
```

```
5     "runtime"
6   )
7
8   func main() {
9     currentMaxProcs := runtime.GOMAXPROCS(0)
10    numCoresAvailable :=  runtime.NumCPU()
11
12    fmt.Printf("Using %d of %d virtual cores.\n", currentMaxProcs,
13        numCoresAvailable)
14
15    runtime.GOMAXPROCS(numCoresAvailable)
16  }
```

Listing 4: Enabling parallelization

The good news is that this is the ONLY difference between having concurrent code and parallel code. No nasty pthread nightmares here.

A Solution

To demonstrate what you might do with Go, I have implemented three methods of matrix multiplication. One is a fairly naive serial algorithm, one is a somewhat naive parallel algorithm, and the third is based off of a cache-optimized algorithm originally intended for C++. All of these functions use a Matrix type which was self-defined. If you're familiar at all with structs in C/C++ then you know basically how to define them. This was for my own convenience, it is absolutely not essential to do.

Serial

```
1   func (A *Matrix) MultiplySerial(B *Matrix) (C *Matrix) {
2     C = ZeroedMatrix(A.rows, B.cols)
3     for i := 0; i < A.rows; i++ {
4       temp := C.elements[row*C.cols : (row+1)*C.cols]
5       for k, a := range A.elements[row*A.cols : (row+1)*A.cols] {
6         for j, b := range B.elements[k*B.cols : (k+1)*B.cols] {
7           temp[j] += a * b
8         }
9       }
10    }
11    return C
12  }
```

There is very little that needs explaining here. It simply ranges across the matrix filling things up as it goes along. The argument to the range functions are examples of performing a 'slice' (hence why they're called slices in the first place). This operation doesn't create new values, just a new wrapper for a subset of original slice.

Parallel Naive

```
1  func (A *Matrix) MultiplyParallelNaive(B *Matrix) (C *Matrix) {
2    C = ZeroedMatrix(A.rows, B.cols)
3    done := make(chan bool)
4    for i := 0; i < A.rows; i++ {
5      go func(i int) {
6        temp := C.elements[row*C.cols : (row+1)*C.cols]
7        for k, a := range A.elements[row*A.cols : (row+1)*A.cols] {
8          for j, b := range B.elements[k*B.cols : (k+1)*B.cols] {
9            temp[j] += a * b
10         }
11       }
12       done <- true
13     }(i)
14   }
15
16   for i := 0; i < A.rows; i++ {
17     <-done
18   }
19   return C
20 }
```

This implementation spawns a goroutine for each row in A.

Parallel Recursive

```
1  func (A *Matrix) MultiplyParallelRecursive(B *Matrix) (C *Matrix) {
2    done := make(chan bool)
3    C = ZeroedMatrix(A.rows, B.cols)
4    go multiplyParallelRecursive(A, B, C, 0, A.rows, 0, B.cols, 0,
5        A.cols, 60, done)
6    <-done
7    return C
8  }
9
10 func multiplyParallelRecursive(A, B, C *Matrix, firstRow, lastRow,
11   firstCol, lastCol, firstK, lastK, threshold int, done chan bool) {
12   di := lastRow - firstRow
13   dj := lastCol - firstCol
14   dk := lastK - firstK
15   done2 := make(chan bool)
16
17   switch {
18   case di >= dj && di >= dk && di >= threshold:
19     middleRow := firstRow + di/2
20     go multiplyParallelRecursive(A, B, C, firstRow, middleRow,
21           firstCol, lastCol, firstK, lastK, threshold, done2)
22     multiplyParallelRecursive(A, B, C, middleRow, lastRow,
23           firstCol, lastCol, firstK, lastK, threshold, nil)
24     <-done2
25   case dj >= dk && dj >= threshold:
26     middleCol := firstCol + dj/2
27     go multiplyParallelRecursive(A, B, C, firstRow, lastRow,
28           firstCol, middleCol, firstK, lastK, threshold, done2)
29     multiplyParallelRecursive(A, B, C, firstRow, lastRow,
30           middleCol, lastCol, firstK, lastK, threshold, nil)
```

```
31      <-done2
32    case dk >= threshold:
33      middleK := firstK + dk/2
34      go multiplyParallelRecursive(A, B, C, firstRow, lastRow,
35            firstCol, lastCol, firstK, middleK, threshold, done2)
36      multiplyParallelRecursive(A, B, C, firstRow, lastRow,
37            firstCol, lastCol, middleK, lastK, threshold, nil)
38      <-done2
39    default:
40      for i := firstRow; i < lastRow; i++ {
41        for j := firstCol; j < lastCol; j++ {
42          for k := firstK; k < lastK; k++ {
43            C.elements[i*C.cols+j] += A.elements[i*A.cols+k]
44                        * B.elements[k*B.cols+j]
45          }
46        }
47      }
48    }
49
50    if done != nil {
51      done <- true
52    }
53  }
```

This implementation was adapted from some C++ code. It works by breaking the matrix up into chunks via recursion until the chunk is under some size threshold. After that threshold, it performs a serial multiplication. Note that in Go it is standard to use a switch statement for any if/else block that has more than two cases.

Benchmarks

I performed the following benchmarks using my own desktop computer, which has a 4-Core Intel i7-4770k processor clocked at 4.1Ghz and is running Arch Linux. The i7 line of processors has 2 virtual cores per logical core. Times are in nanoseconds.

4 Core Trials

Matrix Size	Serial	Naive Parallel	Recursive Parallel
100	99	4016	1317
200	119	2651	1509
300	209	3002	1868
400	99761505	21074647	59863142
500	97807034	41016904	98767981
600	167947010	93872914	179861610
700	164851359	110259828	280103848
800	396063496	172119061	508307211
900	569451181	298691525	528134573
1000	784346706	363710725	803038928

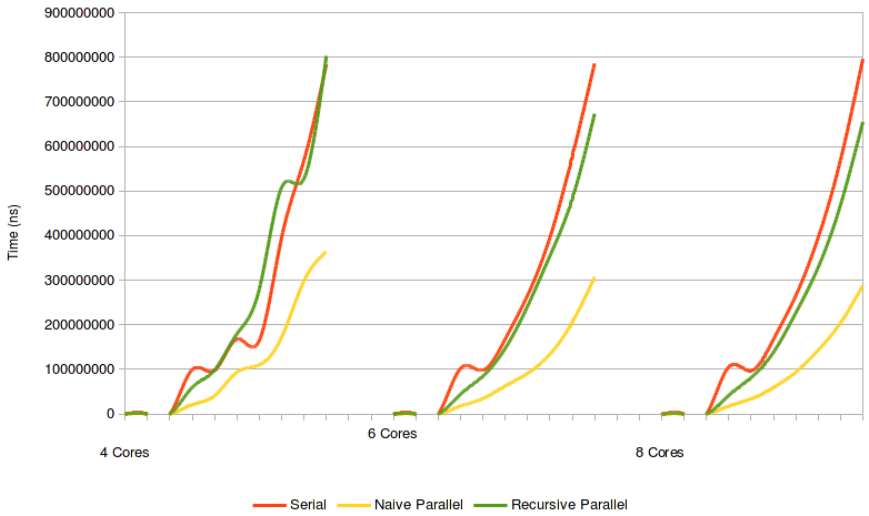
6 Core Trials

Matrix Size	Serial	Naive Parallel	Recursive Parallel
100	98	3785	1312
200	118	2647	1367
300	210	3476	1838
400	102093339	17901939	44367134
500	98436085	34713470	84789085
600	167947010	62855275	146964769
700	265900457	90986734	241514586
800	395576527	134205870	354467118
900	577145442	205746086	482200840
1000	785939321	307003085	673300908

8 Core Trials

Matrix Size	Serial	Naive Parallel	Recursive Parallel
100	99	4365	1308
200	120	1997	1346
300	214	3497	1711
400	105588216	17837782	41711627
500	97397526	33836775	81606909
600	167273770	59226103	138422147
700	265426407	94021277	226854082
800	395498164	143233189	328658993
900	569336575	204324195	468827937
1000	796239772	288327130	654892840

Final Results



The final results are a win for the naive algorithm! Given the complexity of the recursive algorithm, I'm not terribly surprised. The amount of work the naive algorithm has to do to dispactch a goroutine is much less than that of the recursive algorithm. Further, the recursive algorithm spawns far more goroutines than the naive algorithm does. Goroutines may be very cheap, but they certainly are't free.

If you would like to run these benchmarks yourself, drop down into the source folder and run the command 'go test -bench=. -cpu 4,6,8', assuming Go has been installed on your system.