Multi-Core Parallel Programming in Go

Peiyi Tang
Department of Computer Science
University of Arkansas at Little Rock
Little Rock, AR 72204

Abstract

Go is a new concurrent systems programming language. One of its goals is to meet the challenge of multi-core parallel programming. In this paper, we present two multi-core parallel programs in Go and their performances on an octal-core microprocessor, to demonstrate the ease of multi-core parallel programming in Go and the efficiency of parallel Go code.

1 Introduction

Up to early 2000’s, computer architecture was able to hide parallel processing in the hardware, doubling microprocessor performance almost every 18 months without changing the sequential programming model. Due to the increasing gap between processor and memory speeds, the limit of instruction level parallelism as well as high power consumption, all microprocessor manufacturers shifted to multi-core microprocessors in the middle of 2000.

Multi-core microprocessor delivers high performance through multi-processor parallel processing. To get high performance of a job, it has to be programmed in parallel. Running the sequential program of the job only gives the performance of single processor, leaving the resource of other processors on the chip wasted.

The initial response to the multi-core parallel programming is the multi-thread programming [1]. However, writing parallel and concurrent programs in the multi-threading model is extremely difficult. As pointed by Edward A. Lee [2], the problem with programming with threads is that it takes a backward approach, allowing programs to be non-deterministic at the first place and then using semaphores, locks and monitors to prune non-determinism.

Recently, Google announced a new language called Go\(^1\). Go is a concurrent garbage-collected systems programming language [3]. One of its goals is to meet the challenge of multi-core programming to make parallel programming easy. Go does not use the multi-threading model. Instead, it supports concurrency by using Go routine and CSP-like communication channel. Any function in Go can be invoked as a normal routine as in the sequential programming model or a Go routine by using keyword go in front of the routine call. A Go routine is executed concurrently with the calling routine, whether it is run on the same or different processor. From the programmers’ view, Go routine is where concurrency resides, and thus it sets up a clear boundary of non-deterministic operations. Go extends the CSP communication channel [4] with non-zero buffer size to allow asynchronous send (write). Channel is first-class object in Go and can be passed from one routine to another. The combination of Go routine and extended CSP channel provides a powerful mechanism to specify and reason about the concurrent computation with controlled non-determinism.

In this paper, we are going to demonstrate the ease of parallel programming in Go and the efficiency of the multi-core parallel Go code by presenting two programs in Go and their performances. The first program and its performance, presented in Section 2, is for the parallel integration problem which does not have communication or synchronization among parallel tasks. The second program and its performance, presented in 3, is for the parallel dynamic programming problem, which requires synchronization among parallel tasks. Section 4 concludes the paper.

2 Parallel Integration

Calculation of integration is a simple problem often used to demonstrate parallel programming and its speedup on parallel computers. Given a function \(f(x)\), its integration in interval \([a, b]\)

\[
\int_a^b f(x)dx
\]

can be approximated by the summation of the areas of a large number \(n\) of small rectangles under the curve

\(^1\)http://golang.org
where $h = \frac{k-n}{n}$ is the width of the small rectangles. Since there is only a limited number of processors with $np << n$, we only need to create $np$ chunks of computations, one for each processor. To balance the work load, we use the blocking formula from [5] to divide the $n$ rectangles indexed $0, \ldots, n-1$ into $np$ chunks evenly so that the number of rectangles allocated to each chunk differs no more than one. In particular, the chunk $i$ ($i = 0, \ldots, np - 1$) computes the rectangles

$$
\left\lfloor \frac{i \times n}{np} \right\rfloor, \left\lfloor \frac{i \times n}{np} \right\rfloor + 1, \ldots, \left\lfloor \frac{(i+1) \times n}{np} \right\rfloor - 1 \quad (1)
$$

The work to compute a chunk of $\pi$ is done by function `chunk(start, end int, c chan float64)`. It computes the areas of the small rectangles ranged from `start` up to `end`. Channel `c` of `float64` is used both for barrier synchronization at the end of computation and sending the computed chunk to the main routine.

The main routine starts with setting up the runtime system to run Go routines on `np` processors (cores) by calling `runtime.GOMAXPROCS(np)`. The `make()` makes the channel of float64 values `c` with buffer size `np`. The first for loop initiates `np` Go routines executing function `chunk()` and passes the channel `c` to each of them. These Go routines execute concurrently with the main routine. The second for loop executes `pi += <-c` repeatedly `np` times, accumulating in variable `pi` the chunks received from channel `c`. (`<-c` is a unary expression for the value received...
from channel c.) Before the first Go routine running chunk() sends to channel c, it is empty. Receiving from the empty channel c via c = chunk() blocks the main routine. The second for loop completes after it receives np chunks from all the go routines and variable pi will have the correct value then.

We run the code in Figure 1 on an Octal-core AMD Opteron(tm) Processor of 2.8GHz and 1024KB cache. We vary the problem size n from $10^4$ to $10^{10}$ and the number of processors np from 1 to 8. Figure 2 shows the execution time of the parallel Go code calculating $\pi$. It is observed that, for the small problem sizes ($n = 10^4, 10^5$), using more processors increases the execution time. This is because the overhead of initiating and scheduling Go routine dominates the execution time, offsetting the gain by the parallel processing. When problem size n is large ($n = 10^6, 10^9, 10^{10}$), using more processors does decrease the execution time.

To see the speedup of parallel execution using np processors, we calculate the speed as

$$ S_{np} = \frac{T_1}{T_{np}} \tag{2} $$

where $T_1$ and $T_{np}$ is the time using 1 processor and np processors, respectively, and plot them in Figure 3. Note that, when problem size n is large ($n = 10^6, 10^9, 10^{10}$), the speedup is close to the linear speedup. In particular, the speedup of problem size $n = 10^9$ reaches 7.79 when np = 8 processors are used.

### 3 Parallel Dynamic Programming

We now turn to the second multi-core parallel program example for dynamic programming problems. Dynamic programming method is for optimization problems with overlapping subproblems [6]. It takes advantage of overlapping subproblems and computes each subproblem once and stores the solution in a table to be looked up later. It reduces the time complexity from the otherwise exponential to the polynomial. We use the optimal binary search tree problem to show the multi-core parallel code in Go for the dynamic programming methods.

Given n keys, $k_1, \cdots, k_n$, and their probability distribution $p_1, \cdots, p_n$, the optimal binary search tree problem is to find the binary search tree (BST) of these keys with minimum average search time.

Let $BST_{i,j}$ denote the optimal BST containing keys, $k_i, \cdots, k_j$ ($j \geq i-1$), and $MST_{i,j}$ its mean search time (MST). $BST_{i,i}$ is a single node tree with root $k_i$ ($1 \leq i \leq n$) and its mean search $MST_{i,i}$ is $p_i$. Also $BST_{i,i-1}$ is an empty tree ($1 \leq i \leq n$) and this mean search time is $MST_{i,i-1} = 0$.

```go
func mst(i,j int) {
  var bestCost float = 1e9 + 0.0
  var bestRoot int = -1
  switch {
    case i > j:
      cost[i][j] = 0.0
      root[i][j] = -1
    case i+1 == j:
      cost[i][j] = prob[i]
      root[i][j] = i+1
    case i+1 < j:
      psum := 0.0
      for k := i; k <= j-1; k++ {
        psum += prob[k]
      }
      for r := i; r <= j-1; r++ {
        rcost := cost[i][r] + cost[r+1][j]
        if rcost < bestCost {
          bestCost = rcost
          bestRoot = r+1
        }
      }
      cost[i][j] = bestCost + psum
      root[i][j] = bestRoot
  }
}
```

![Figure 4: Dynamic Programming Algorithm](image)

If the optimal binary search tree for $k_1, \cdots, k_n$ has $k_r$ ($1 \leq r \leq n$) as its root, then its left sub-tree containing $k_1, \cdots, k_{r-1}$ and right sub-tree containing $k_{r+1}, \cdots, k_n$ must also be optimal. Therefore, $MST_{i,j}$ can be defined recursively as follows:

$$ MST_{i,j} = \min_{i \leq r \leq j} (MST_{i,r-1} + MST_{r+1,j}) + \sum_{k=i}^{j} p_k \tag{3} $$

The value of r that gives the minimum of the sums $MST_{i,r-1} + MST_{r+1,j}$ determines $k_r$ as the root of $BST_{i,j}$. 
The data structure to store $MST_{i,j}$ is the upper triangular sub-array of an $(n+1) \times (n+1)$ matrix $cost[n+1][n+1]$. In particular, $MST_{i,j}$ is stored in $cost[i+1][j]$. Similarly, the root of the optimal binary search tree containing $k_1, \cdots, k_n$ is stored in $root[i-1][j]$ of another matrix $root[n+1][n+1]$. The probability distribution of keys is stored in an array $prob[n]$ with $p_i$ in $prob[i-1]$.

$$
\text{cost}[i-1][j] \\
\text{cost}[i][j-1] \rightarrow \text{cost}[i][j] \rightarrow \text{cost}[i][j+1] \\
\text{cost}[i+1][j]
$$

Figure 5: Data Dependencies between Tasks

The sequential dynamic programming algorithm to find the optimal binary search tree [7] coded in Go is shown in Figure 4. Basically, the algorithm computes $cost[i][j]$ ($0 \leq i \leq j \leq n$) using the values in $cost[i][i], cost[i][i+1], \cdots, cost[i][j-1]$ and the values in $cost[i+1][j], cost[i+2][j], \cdots, cost[j][j]$. There are data dependencies from the tasks computing $cost[i][j-1]$ and $cost[i+1][j]$ to the task computing $cost[i][j]$. Figure 5 show such data dependencies involving $cost[i][j]$. This is the reason that the nested loop in the sequential algorithm in Figure 4 follows the order of bottom-up and left-to-right in computing $cost[i][j]$. When parallelizing the algorithm, we must honor these data dependencies.

$$
\text{chunk}(i,j \text{ int}) \\
\text{if}(i < j) \\
\text{else}
$$

Figure 6: Task Space Tiling and Channels

In principle, we can create as many as $(n+1)(n+2)/2$ parallel tasks, one for computing each $cost[i][j]$ ($0 \leq i \leq j \leq n$). But, the granularity of the parallel tasks would be too small and the scheduling and synchronization cost could be large enough to offset the gain of the parallel computing. To control the granularity of the parallel computing, we partition the task space into $vp(vp+1)/2$ tiles and allocate each tile to a Go routine as a unit of computation. Within each tile, the Go routine computes the portion of the arrays $cost$ and $root$ in the bottom-up and left-to-right order. In order to enforce the data dependencies between tiles induced by the data dependencies shown in Figure 5, we use channels for synchronization.

Figure 6 shows the 10 tiles with $vp = 4$ of the problem of size $n = 11$ and Go channels for synchronization. Each dot represents the computing of an element of array $cost$ and called a point. The arrows between tiles are Go channels for synchronization. In particular, a tile can start its computation only after the tiles below and on the left are completed. Each tile has an index $(i,j)$ ($0 \leq i \leq j \leq vp - 1$). Channels are identified with the task which sends to them. In particular, tile $(i,j)$ transmits signal through horizontal channel $h_{i,j}$ to the tile $(i,j+1)$ at its right (if it is not at the right border (i.e. $j < vp - 1$)), and through vertical channel $v_{i,j}$ to the tile $(i+1,j)$ above (if it is not at the top border (i.e. $i > 0$)). The tiles $(i,j)$ on the diagonal $(i = j)$ can start their computation at the beginning as they do not depend on any other tiles. The top-right tile $(vp-1,vp-1)$ is the last one to compute. To signify the completion of the last tile, we have another channel called $finish$, through which it sends signal to the main routine.

The abridged parallel Go code for the optimal binary search tree is shown in Figure 7. The main routine first creates all the horizontal and vertical channels plus channel finish. The computation for a tile is specified by function $\text{chunk}(i,j \text{ int})$, where $i$ and $j$ are the index of the tile. The main routine then initiates $vp(vp+1)/2$ Go routines for all the tiles. Note the order of Go routine initiation: we start the tiles on the diagonal $j = i$ first, then those on the sub-diagonal $j = i + 1$, and so on. The reason is that the tiles on the diagonal can start on multiple processors as early as possible. The main routine then waits on receiving a signal from channel finish. The function $\text{chunk}(i,j \text{ int})$ first calculates the index ranges in both dimensions using the blocking formula in (1). All the tiles not on the diagonal $(i < j)$ need to wait for the completion of the tiles on its left and below. It computes the points in the tile by calling the same function $\text{mst}()$ as in the sequential code (see Figure 4). Afterward, the tile needs to send signal through channel $h_{i,j}$ ($v_{i,j}$) to the tile on its right (above), if it is not on the right (top) border. The last tile $(i,j) = (0,vp-1)$ sends a signal through channel $finish$ to the main routine,
```go
var h, v [vp][vp] chan int
var finish chan int
// declare other variables and constants
func creatChan() {
    for i := 0; i < vp; i++ {
        for j := i; j < vp; j++ {
            if j < vp-1 {h[i][j] = make(chan int, 1)}
            if i > 0 {v[i][j] = make(chan int, 1)}
        }
    }
}

func mst(i, j int) {
    // the same as in the sequential code
}

func chunk(i, j int) {
    var bb int
    il := (i * (n+1))/vp // block-low for i
    ih := ((i+1) * (n+1))/vp - 1 // block-high for i
    jl := (j * (n+1))/vp // block-low for j
    jh := ((j+1) * (n+1))/vp - 1 // block-high for j
    if i < j { // not a tile on the diagonal
        <-h[i][j-1] // receive from the left
    }
    <-v[i+1][j] // receive from below
    for ii := il; ii >= il; ii-- {
        if i == j { bb = ii } else { bb = jl }
        for jj := bb; jj <= jh; jj++ {
            mst(ii, jj)
        }
    }
    if j < vp-1 {// not a tile on the right border
        h[i][j] <- 1
    }
    if i > 0 {// not a tile on the top border
        v[i][j] <- 1
    }
    if i == 0 && j == vp-1 {// the last tile
        finish <- 1
    }
}

func main() {
    // read flags
    runtime.GOMAXPROCS(np)
    // start timing
    creatChan()
    finish = make(chan int, 1)
    for d := 0; d < vp; d++ { // sub-diagonal of j=i+d
        for i := 0; i+d < vp; i++ {
            go chunk(i, i+d)
        }
    }
    <-finish
    // end timing
}
```

Figure 7: Parallel Dynamic Programming in Go

Figure 8: Execution Time of Problem Size n=2000

We chose the problem size $n = 2000$ for the experiments so that the execution time is large enough for accurate measurement. We run the parallel code in Figure 7 on the Octal-core AMD Opteron(tm) microprocessor, varying the number of tile blocks in each dimension $vp$ from 1, 2, 4, 8, · · · , 1024 and 2001. The larger $vp$, the smaller the granularity. $vp = 1$ corresponds to the largest granularity with only one tile and no parallelism. It has only one Go routine call and one channel finish. Its execution time is almost the same as the sequential code in Figure 4 no matter how many processors are used. $vp = 2001$ corresponds to the smallest granularity with largest number of tiles (2,003,001), one for each point. Figure 8 shows the parallel execution time of the problem for different $vp$ using different number of processors, $np$. For each configuration of $vp$ and $np$, we run the program 5 times in the single-user mode and calculate the average before plot it in Figure 8. Note the execution time for $np = 1$. We observe that the execution time decreases as $vp$ increases from 1 to 32. This is because increasing $vp$ creates more tiles and increases the data locality for cache. When $vp$ further increases from 32 to 2001, the execution time increases because the data locality diminishes and the overhead of creating and scheduling Go routines increases. We run another sequential but tiled code (the code is not shown) obtained by removing all channels and changing Go routine call to normal routine call in the parallel code in Figure 7. Its execution time is plotted as "seq-tile" in Figure 8. This time demonstrates the true performance gain by the cache data locality due to the tiling without Go routine overhead. Therefore, the difference between "seq-tile" and "np=1" is the overhead of Go routine calls and their scheduling. For $vp = 2001$, which has 2,003,001 Go routine calls, the
The difference between "np=1" and "seq-tile" is 62.318429-43.351857=18.966572 seconds. Thus, the cost of create and scheduling a Go routine in average when one processor is used is 18.966572/2003001 = 9.46 * 10^{-6} seconds or 9.46 \mu s.

The parallel execution time on multi-cores for np > 1 are all less than the corresponding time for np = 1. However, the time for "np=1" contains the overhead of Go routine calls and their scheduling which are not truly needed in the sequential code, we use the tiled sequential time of "seq-time" as the base to calculate the speedup of parallel multi-core computing. In particular, the speedup of using np ≥ 1 processors is

\[ S_{np} = \frac{T_{ts}}{T_{np}} \]  

where \( T_{ts} \) and \( T_{np} \) are the tiled sequential time and the parallel time using np processors, respectively. Figure 9 shows the speedup of parallel execution time over the tiled sequential time for variety of tile sizes. The best speedup is achieved when vp = 128. For 8 processors (np = 8), the speedup is 7.70 when vp = 128. As vp increases from 128, the granularity decreases and the overhead of Go routines calls and their scheduling slows down the execution, reducing the speedup as shown in Figure 9(a). As vp decreases from 128, the granularity increases and there are less tiles, thus less parallelism, also reducing the speedup. Figure 9(b) shows the reduced speedup due to the reduced parallelism.

4 Conclusion

We have presented the Go parallel codes for two different parallel computing problems: parallel integration and parallel dynamic programming. These codes show the ease of writing multi-core parallel programs using Go. We also measure the performance of the codes. The highest speedups of parallel integration and parallel dynamic programming on an octal-core AMD chip (8 processors) were 7.79 and 7.70, respectively. The cost of initiating and scheduling a Go routine when using one processor is as low as 9.46 \mu s. Given that parallel computing tends to run large jobs, the overhead of Go routine of this magnitude should be considered very small.

References