M-CS259-Analysis of Parallel Implementation of algorithms in Python and Golang

Jayant Shelke
Computer Science Department
San Jose State University
San Jose, CA 95192
408-478-5218
Jayant.shelke@sjsu.edu

ABSTRACT

In recent years parallel processing and distributed computing have become really prominent in importance due to their advantages of scalability and effective throughput utilization. It is important to mention, throughput can be of two types. Inter machine connected architecture throughput and intra machine performance throughput. Distributed computing deals with inter machine throughput whereas parallel processing deals with intra machine throughput. With the ever-increasing demands of scaling and high availability, it has become imperative that we focus on both fronts and try to improve them as much as possible. In this study, we are going to take a magnified look at how a particular algorithm can be parallelized and how it can be executed most efficiently. Also, we will compare the performance in relation with execution times of the same parallel implementation of a sequential sorting algorithm in 2 languages – Python and Golang. This shall give us a clear understanding of some key consideration that we need to take care of while implementing algorithms for parallel processing. Many different algorithms could be used but the more suitable candidates for parallelization of algorithms are the ones which have divide and conquer approach to their solution. Thus, we select Quicksort. Merge sort could also be an equally good fit but, since, quicksort performs in quadratic time in worst case, it was a more potent choice to see if the parallelization would yield more fruitful results.

1. INTRODUCTION

With the ever-increasing demand for speed of service along with unlimited and unchecked increase in size of data comes the need to more and more efficient ways of handling these requests. Today, the amount of data generated by google searched is huge [1]. Google serves roughly 40,000 searches per second. This makes it a staggering 3.5 billion searches per day [1]. There are many websites out there where information search and retrieval is a really challenging task simply because of the size of data being queried against. With the increase in network access speeds, hardware performance and scalable distributed architectures, there is still a core factor that controls the actual speed to execution. This factor is the programmatic design of the solution. This follows algorithms at its base and after a certain level of optimization, the only option left if to switch from sequential to parallel processing. Parallel processing is essentially the ability of employing more people to do the same job to get it done faster. However, it is far more difficult than it was expressed in the previous statement to achieve performance gains from parallelization. It involves intricate changes to the implementation, careful understanding of the architecture of the programming language used and a deep study of how the operating system behaves in terms of memory, process and thread worker management. In the following section, we will discuss some key terms and concepts that are essential for this paper and also explain the sequential and parallel approaches to Quicksort algorithm that we are considering for parallelization.

2. Ontology

Before heading to the actual implementation and results, we define certain concepts which are crucial for the correct conceptual reflection of this paper.

2.1 Concepts

We discuss the concepts of Operating system that are relevant for parallel processing.

2.1.1 Process

A Process in an operating system is a basic body encompassing execution. Processes not only have their own memory but have minimum and maximum working set sizes, a priority classification, environment variables, unique identifiers, security contexts, handles to system objects, executable code and virtual address spaces. Each process has at least one thread, primary thread but is not limited to and can create more threads under the same memory space allocated to the process. If there was no thread in a process, it would just be process state and program code loaded in memory and would not execute. This is analogous to a road with no vehicles. Thus, a process is an executing instance of a program.

2.1.2 Thread

A Thread is an entity within a process that can be scheduled for execution. Each thread created by a process will share the same memory of the process and its system resources. Each thread will have a scheduling priority, thread local storage, thread identifier and a set of structures the system will use to store the state of the thread until it is scheduled. Aa per the above analogy, thread is a vehicle on the road while the road is a process. However, this is not to be confused with vehicles changing roads. Threads are encompassed by processes and so threads don't get transferred to other processes. Context switching and transferring is however possible which is known as context hand-off between threads across different processes. Multiple threads of control can exploit the true parallelism possible in multi processes systems.

2.1.3 Concurrency

Concurrency refers to the concept of two or more tasks which start, run and stop in overlapping time periods and in no specific order.

This is made possible through 'time-slicing' feature of the operating system. Based on priority of tasks, operating system assigns CPU to each task and allows them to complete execution. This switching is so fast that the end user feels like all tasks are running in parallel. This is known as Concurrency. Concurrency is dealing with multiple of things at once.

2.1.4 Parallelism

Parallelism does not require two tasks to exist. It literally runs parts of a task or multiple tasks at the same time using multiple cores of the system. Each task or part of a task is executed in a different core. This is true parallelism since there is no time slicing happening. Both cores are running the work assigned to them at the exact time instance. This is known as Parallelism. Parallelism is doing multiple things at once. Concurrency is composition of independent processes while parallelism is the simultaneous execution of computations.

2.1.5 Divide and Conquer

This is an approach of solving algorithmic problems. In Contrast to greedy approach, divide and conquer technique divides the main problem into sub problems and tries to solve the smaller problems first. The solution of the smaller problems then constitutes the solution to the bigger problem. This technique is especially useful in getting optimal solutions to sorting problems.

2.1.6 GoRoutines

Golang is a statically typed compiled programming language. A distinct feature of Golang is are the GoRoutines. A GoRoutine is a lightweight thread managed by the Go Runtime. The cost of creating a GoRoutine is tiny as compared to a thread which is why it is common to have thousands of GoRoutines running in a program execution. We shall see in the results why GoRoutines are distinctively different from conventional threads.

2.1.7 Global Interpreter lock [GIL]

GIL is a mechanism in computer language interpreters to coordinate and synchronize the execution of threads and to enforce only one native thread execute at one time. An interpreter which uses a GIL allows only one thread to execute at one time even if running on a multi core processer.

2.1.8 Quicksort algorithm

Quicksort is sorting algorithm just like merge sort. The idea is to pick a pivot element and partition the collection of elements around this pivot element. There are many different approaches to picking the pivot element. However, the most crucial step is the partition step. In partition, for a collection of elements and an element x which is pivot, we need to arrange x at its correct position such that all elements before x are lower than x and all elements after x are greater than x. This should be done in linear time.

Pseudocode for Quicksort algorithm is as follows:

Partition Function:

```
Step 1 – Choose the highest index value as
      pivot
Step 2 – Take two variables to point left
       and right of the list excluding pivot
Step 3 – left points to the low index
Step 4 – right points to the high index
Step 5 – while value at left is less than
       pivot move right
Step 6 – while value at right is greater
       than pivot move left
Step 7 – if both step 5 and step 6 does not
       match swap left and right
Step 8 – if left ≥ right, the point where
       they met is new pivot
```

Quicksort Function:

```
Step 1 – Make the right-most index value
       pivot
Step 2 – partition the array using pivot
       value
Step 3 – quicksort left partition
       recursively
Step 4 – quicksort right partition
       recursively
Step 5 – Once all inner quicksorts are
       returned the final collection is
       automatically sorted.
```

3. Setup

3.1 Problem Definition:

Now, that the algorithm choice is made clear, the problem is setup as follows. We use a collection of integers of incremental lengths which are initialized randomly. These randomly initialized collection of elements or programmatically known as arrays will be used to perform quicksort sequentially as well as using parallel implementation. Both the sequential and parallel implementations will be benchmarked and also in 2 languages of our choice – Python 3.7 and Golang. These benchmarking results will then be compared to infer which language has better support for parallel implementation and the causes of the result shall be discussed.

3.2 Infrastructure:

For this problem benchmark:

Machine - Macbook pro laptop

Operating system - macOS Mojave version 10.14

Processor - 2.3 GHz Intel Core i5

Memory - 8 GB 2133 MHz LPDDR3

Graphics memory - Intel Iris Plus Graphics 640 1536 MB

3.3 Code:

3.3.1 Python Implementation:

The following code modules were used to create the benchmark for quicksort algorithm in sequential and parallel mode. The input array started from an array of 1000 randomly generated integers and increased in size up to a maximum of 15million random integers. The iterations were increases by a margin of 150,000 elements per iteration. Thus, we had 100 iterations on sequential and parallel quicksort benchmarking with the input array of first iteration being 1000 and the last iteration being 15 million.

Below is the main implementation of sequential and parallel Quicksort. There rest of the code is attached with the paper.

```
def quicksort(lst):

  less = []
  pivotList = []
  more = []
  if len(lst) <= 1:
     return lst
  else:
     pivot = lst[0]
     for i in lst:
        if i < pivot:
           less.append(i)
        elif i > pivot:
           more.append(i)
        else:
           pivotList.append(i)
     less = quicksort(less)
     more = quicksort(more)
     return less + pivotList + more

def quicksortParallel(lst, conn, procNum):
  less = []
  pivotList = []
  more = []

  if procNum <= 0 or len(lst) <= 1:
     conn.send(quicksort(lst))
     conn.close()
     return
  else:
     pivot = lst[0]
     for i in lst:
        if i < pivot:
           less.append(i)
        elif i > pivot:
           more.append(i)
        else:
           pivotList.append(i)

  pconnLeft, cconnLeft = Pipe()
  leftProc = Process(target=quicksortParallel,
                args=(less, cconnLeft, procNum - 1))
  pconnRight, cconnRight = Pipe()
  rightProc = Process(target=quicksortParallel,
                args=(more, cconnRight, procNum - 1))
  leftProc.start()
  rightProc.start()
  conn.send(pconnLeft.recv()+pivotList + pconnRight.recv())
  conn.close()
  leftProc.join()
  rightProc.join()
```

3.3.2 *Golang Implementation:*

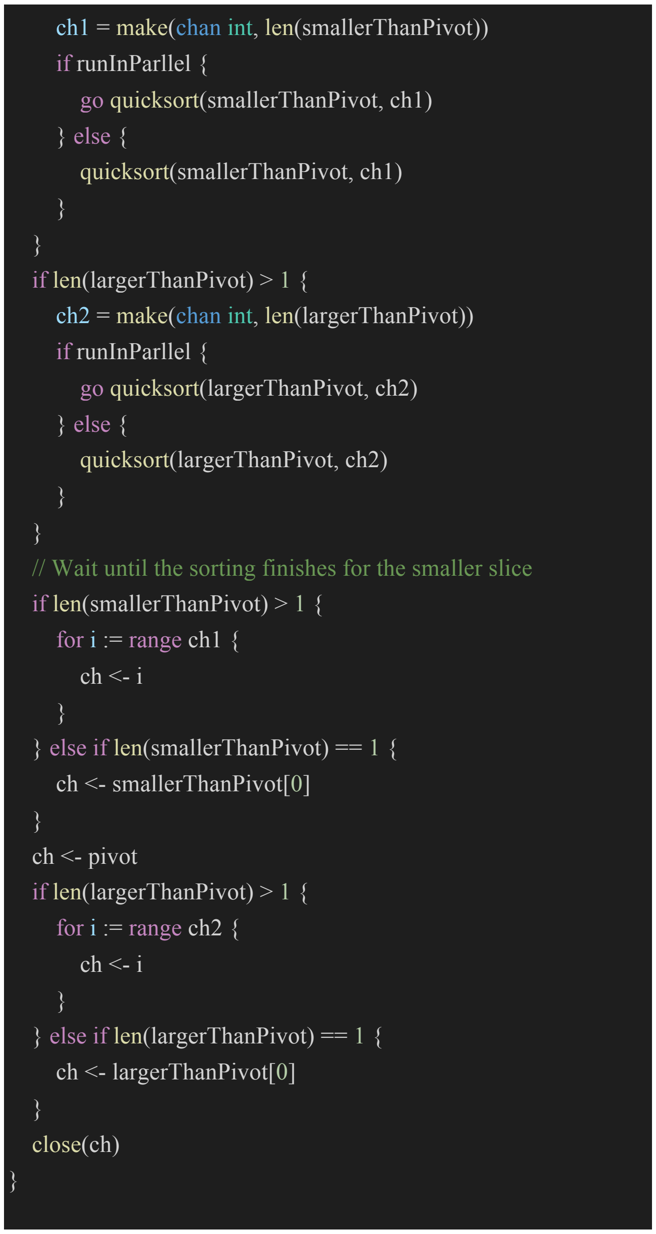
Golang code used the similar idea of using incremental iterations with an increment of 150,000 integers per iteration. However, Golang's final iteration has a much larger array. It uses 40 million integers for sequential and parallel sorting in its final iteration. This is roughly thrice of what python code was able to use.

Below is the main chunk of implementation:

```
var runInParllel bool

func Quicksort(nums []int, parallel bool) ([]int, time.Duration) {
   started := time.Now()
   ch := make(chan int)
   runInParllel = parallel
   go quicksort(nums, ch)
   sorted := make([]int, len(nums))
   i := 0
   for next := range ch {
      sorted[i] = next
      i++
   }
   return sorted, time.Since(started)
}

func quicksort(nums []int, ch chan int) {
   // Choose first number as pivot
   pivot := nums[0]
   // Prepare secondary slices
   smallerThanPivot := make([]int, 0)
   largerThanPivot := make([]int, 0)
   // Slice except pivot
   nums = nums[1:]
   // Go over slice and sort
   for _, i := range nums {
      switch {
      case i <= pivot:
         smallerThanPivot = append(smallerThanPivot, i)
      case i > pivot:
         largerThanPivot = append(largerThanPivot, i)
      }
   }
   var ch1 chan int
   var ch2 chan int
   // Now do the same for the two slices
   if len(smallerThanPivot) > 1 {
```



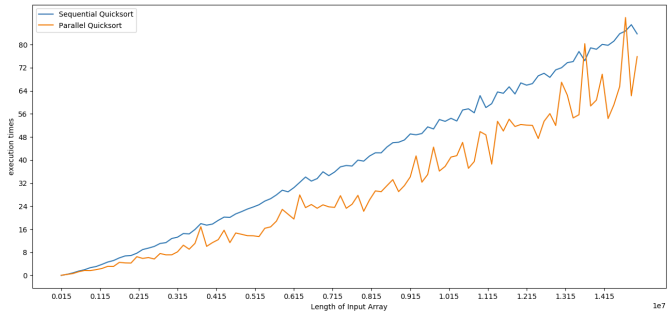
```
        ch1 = make(chan int, len(smallerThanPivot))
        if runInParllel {
            go quicksort(smallerThanPivot, ch1)
        } else {
            quicksort(smallerThanPivot, ch1)
        }
    }
    if len(largerThanPivot) > 1 {
        ch2 = make(chan int, len(largerThanPivot))
        if runInParllel {
            go quicksort(largerThanPivot, ch2)
        } else {
            quicksort(largerThanPivot, ch2)
        }
    }
    // Wait until the sorting finishes for the smaller slice
    if len(smallerThanPivot) > 1 {
        for i := range ch1 {
            ch <- i
        }
    } else if len(smallerThanPivot) == 1 {
        ch <- smallerThanPivot[0]
    }
    ch <- pivot
    if len(largerThanPivot) > 1 {
        for i := range ch2 {
            ch <- i
        }
    } else if len(largerThanPivot) == 1 {
        ch <- largerThanPivot[0]
    }
    close(ch)
}
```

The complete code of both the implementation is attached with the paper for perusal.

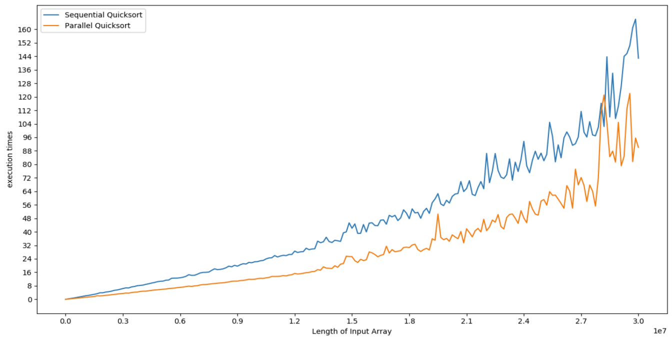
3.4 Results

The results of the implementations are as follows:

Python results:



Golang results:



3.4.1 Execution Times

In the above benchmarking results for quicksort algorithm, it can be clearly seen that the parallel implementation is performing better than the sequential implementation by quite a margin for python. The parallel benchmark is consistently lower than the sequential. The spikes that we see in parallel at the end of the curve for python are due to the unknown behavior of thread pool in swapping and changing execution contexts.

On the other hand, in above benchmarking for Golang, it is clearly evident that the parallel implementation of quicksort in Golang is not only consistently better performing than sequential but it is also relatively more stable in terms of execution context than Python. There are relatively less spikes in execution times as compared to the python implementation.

Something worth noticing is that the even though the Golang graph has more points on the x and y axis, the performance is actually way better than python. Here some exact stats which are not clearly visible in the graphs. If we look at the maximum array size processed by Python, it is 15 million rows. For 15 million rows, the sequential and parallel implementation take 83 and 75 seconds respectively. The same stats can be seen in the table as follows:

Table 1. Execution times in seconds for Python Implementation.

Array Size	Sequential	Parallel
14101000	80.10243607	69.76002526
14251000	79.77317286	54.38516402
14401000	81.21993303	59.15694118
14551000	83.78555369	65.48339295
14701000	84.74451375	89.39960504
14851000	86.92747903	62.27946281
15001000	83.73421812	75.85895824

However, if we look at the stats for same array sizes in Golang implementation, there is a clear distinction in the execution times. Stats for Golang are below:

Table 2. Execution times in seconds for Golang Implementation.

Array Size	Sequential	Parallel
14101000	34.98889871	19.87778414
14251000	34.69060841	18.96995939
14401000	34.33038935	20.83327638
14551000	39.42464091	21.1756795
14701000	39.96576315	25.5739219
14851000	45.31613237	25.33994799
15001000	42.12062131	25.29869407

We can see that the execution times for sequential implementation is less than half the time for python. Also, the parallel implementation for Golang takes only 1/3rd of the time taken by python implementation. This is a massive improvement.

3.4.2 *Capacity*

Also, if we look at the above stats, we clearly see that the amount of data that is handled by the individual implementations is different and for a clear reason. Golang is able to handle a lot more data at execution times still less than the highest execution time of python which is at a lower data capacity. This shows that Golang is better at handling concurrent processing as compared to python.

The maximum capacity of array size for python used was 15 million with the stats mentioned above. Golang, because of its efficient use of memory and swapping strategy, is able to handle up to 30 million records of data and the execution times do not rise proportionally. They are still comparatively low. The maximum array size and stats for Golang implementation are as follow:

Table 3. Execution times in seconds for Golang Implementation with higher array size.

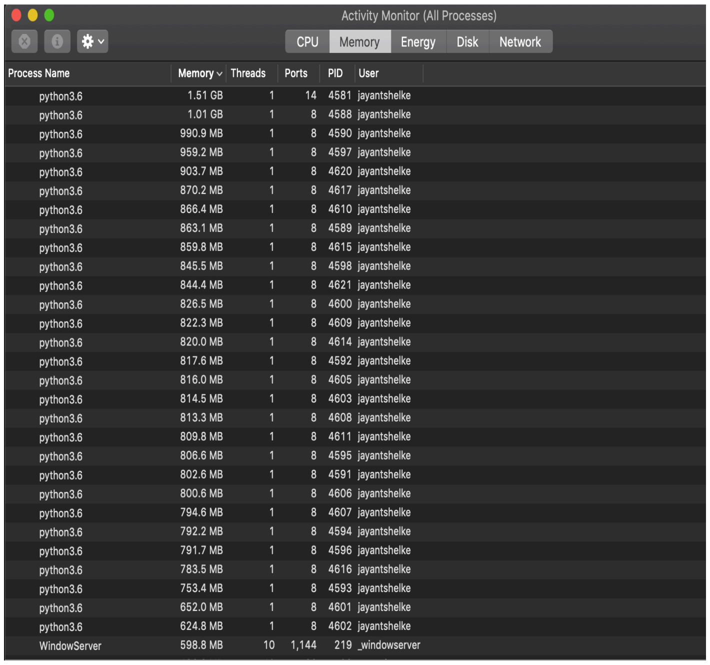
Array Size	Sequential	Parallel
29101000	126.345428	79.11826823
29251000	143.9976069	84.75671245
29401000	145.6901689	113.062125
29551000	150.3979453	122.0205745
29701000	160.9845898	81.60352512
29851000	166.0208907	95.58078759
30001000	142.8786669	90.02659526

From the above data, it can be seen that, Golang's execution time is the same as Python's execution time for almost twice as much data in parallel implementations. [Comparison on Table 1 and Table 3 Parallel Columns.]

3.4.3 *CPU Usage*

When running the computations, one of the important characteristics to be tested is the CPU and Swap memory usage.

Below are the stats for CPU and Swap memory usage from the Python parallel implementations.

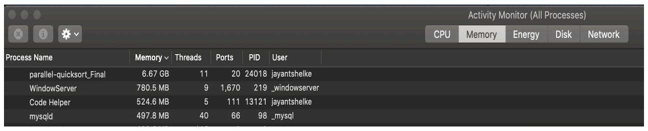


Process Name	Memory	Threads	Ports	PID	User
python3.6	1.51 GB	1	14	4581	jayantshelke
python3.6	1.01 GB	1	8	4588	jayantshelke
python3.6	990.9 MB	1	8	4590	jayantshelke
python3.6	959.2 MB	1	8	4597	jayantshelke
python3.6	903.7 MB	1	8	4620	jayantshelke
python3.6	870.2 MB	1	8	4617	jayantshelke
python3.6	866.4 MB	1	8	4610	jayantshelke
python3.6	863.1 MB	1	8	4589	jayantshelke
python3.6	859.8 MB	1	8	4615	jayantshelke
python3.6	845.5 MB	1	8	4598	jayantshelke
python3.6	844.4 MB	1	8	4621	jayantshelke
python3.6	826.5 MB	1	8	4600	jayantshelke
python3.6	822.3 MB	1	8	4609	jayantshelke
python3.6	820.0 MB	1	8	4614	jayantshelke
python3.6	817.6 MB	1	8	4592	jayantshelke
python3.6	816.0 MB	1	8	4605	jayantshelke
python3.6	814.5 MB	1	8	4603	jayantshelke
python3.6	813.3 MB	1	8	4608	jayantshelke
python3.6	809.8 MB	1	8	4611	jayantshelke
python3.6	806.6 MB	1	8	4595	jayantshelke
python3.6	802.6 MB	1	8	4591	jayantshelke
python3.6	800.6 MB	1	8	4606	jayantshelke
python3.6	794.6 MB	1	8	4607	jayantshelke
python3.6	792.2 MB	1	8	4594	jayantshelke
python3.6	791.7 MB	1	8	4596	jayantshelke
python3.6	783.5 MB	1	8	4616	jayantshelke
python3.6	753.4 MB	1	8	4593	jayantshelke
python3.6	652.0 MB	1	8	4601	jayantshelke
python3.6	624.8 MB	1	8	4602	jayantshelke
WindowServer	598.8 MB	10	1,144	219	_windowserver

It is important to understand that this screenshot does not signify the total RAM of the machine. This screenshot tells the total amount of memory swapping done by each thread that was started by the python program. This is one of the reason why python parallel implementation started getting slow after 10 million rows and went on becoming exponentially slow on account of using more and more swap memory and thus causing a lot of context and switching overhead.

On the other hand, the Golang implementation has a memory model which does far more efficient use of the memory. It essentially still uses quite a lot memory but does not slow down other processes due to the way GoRoutines are implemented. They are self-contained light-weight threads which are executed as part of the routine and so they don't start multiple threads like python implementation. All the routines are run within the same process and so memory sharing is easier and does not cause a lot of context switches.

Below are the stats for Golang Implementation. The first program shown in the image is the Golang implementation.



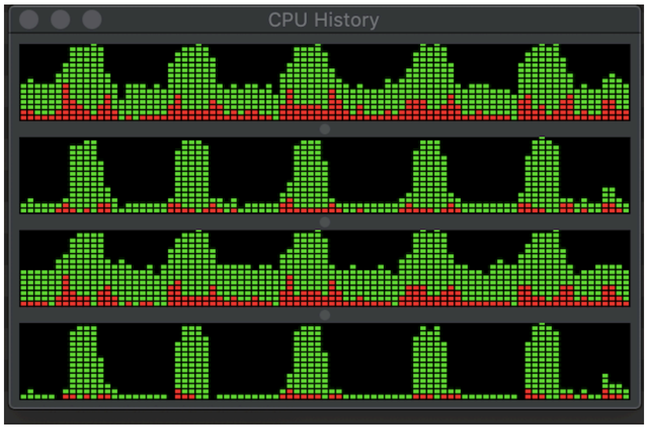
Process Name	Memory	Threads	Ports	PID	User
parallel-quicksort_Final	6.67 GB	11	20	24018	jayantshelke
WindowServer	780.5 MB	9	1,670	219	_windowserver
Code Helper	524.6 MB	5	111	13121	jayantshelke
mysqld	497.8 MB	40	66	98	_mysql

3.4.4 *Core Utilization*

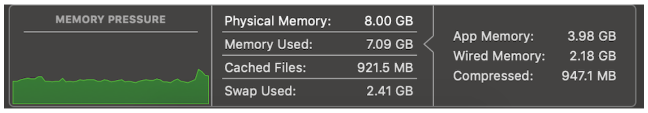
Python parallel implementation essentially is not utilizing all the cores because Python's interpreter is governed by the Global Interpreter Lock[GIL]. GIL can reduce the performance of multi-threaded code in Python. The net effect, in most cases, is that threads can't actually run simultaneously because of locking contention.

However, Golang's concurrency and parallelization has no such restriction and so it can use all possible cores for execution. Golang has a runtime variable knows as `GOMAXPROCS` which can be set to twice the number of CPU's to be able to best utilize the available cores for execution.

Below is a snapshot of memory and CPU core usage for the Golang parallel implementation. The workload is utilizing all cores and the swap memory used is really low as compared to Python where it was in the bound of 10gb for half the data size. The machine used had 4 cores and so the image has 4 partitions.



Memory usage stats:



Physical Memory:	8.00 GB
Memory Used:	7.09 GB
Cached Files:	921.5 MB
Swap Used:	2.41 GB
App Memory:	3.98 GB
Wired Memory:	2.18 GB
Compressed:	947.1 MB

It is worth mentioning why the Golang's GoRoutines perform better than python's thread or processes since, GoRoutine are nothing but light-weight threads [3].

Some of the prime reasons why this is the case are:

- Goroutines are extremely cheap when compared to threads with only a few kb in stack size and stack can grow and shrink according to needs of the application whereas in the case of threads the stack size has to be specified and is fixed.
- GoRoutines are multiplexed to fewer number of OS threads. There might be only one thread in a program with thousands of GoRoutines. If any GoRoutine in that thread blocks say waiting for user input, then another OS thread is created and the remaining GoRoutines are moved to the new OS thread. All these are taken care by the runtime.
- GoRoutines communicate using channels. Channels by design prevent race conditions from happening when accessing shared memory using GoRoutines.

4. CONCLUSION

In conclusion, we see that the parallel implementation of an algorithm has many factors that can influence its performance. From the comparison between Python and Golang implementations, we saw that Golang was way better suited for an algorithm which should naturally be parallelized and is an easy choice for parallel programming. Python definitely has modules that guide the work flow for multiprocessing and multithreading but due the Global interpreter lock, python code is not able to scale and perform the same as Golang. There may be some tweaks in the python implementation where we manage the context switching and memory swaps manually and make it perform as good as Golang. However, it's a manual effort and not built in to the language base.

5. REFERENCES

[1] http://www.internetlivestats.com/google-search-statistics/
Data about the amount of google searched per day.

[2] https://www.geeksforgeeks.org/quick-sort/

[3] https://golangbot.com/goroutines/

[4] https://golang.org/ref/spec

[5] Horacio Gonzalez-Velez, Mario Leyton, A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers **https://doi.org/10.1002/spe.1026**

[6] H. E. Bal, R. van Renesse, A. S. Tanenbaum, "Implementing distributed algorithms using remote procedure calls", Proc. AFIPS Nat. Computer Conf., pp. 499-506, 1987-June.

[7] K. Li, "IVY: a Shared Virtual Memory system for parallel computing", *Proc. 1988 Int. Conf. Parallel Process*, pp. 94-101, 1988-Aug.