Abstract
Programming parallel machines as adequately as successive ones would preferentially require a language that provides high-level programming constructs to avoid the programming errors which are frequent while communicating in parallel environment. In as much as undertaking parallelism is regularly viewed as more error-prone than data parallelism, we survey three mainstream and productive parallel programming languages that handle this troublesome issue: OpenMP, CUDA, and GO. Using examples of parallel implementation that handle synchronization, this paper depicts how the fundamentals of parallel programming, namely collective and point-to-point synchronization, are managed in these languages. Our study proposes that, despite the fact that there is an abundance of different names and thoughts presented by these Languages, the paper is intended to give users and designers (of language and compiler) a diagram of current parallel languages.

Keywords: Parallel programming languages, Synchronization, OpenMP, CUDA, and GO.

Table of Contents
1. Introduction
2. Problem Statement (Parallel Programming Languages)
   2.1 OpenMP
   2.2 CUDA
   2.3 GO
3. Proposed Approach and Outcomes
4. Conclusion
One Page Critique
References
1. Introduction
The meaning of parallel programming language that is an official explanation for algorithm expression can be characterized by appealing to a parallel computational model. The sequential languages have a simple data model that there is a single address space of memory locations that can be read and written by the processor, this is known as the Random Access Memory (RAM) model. However, in the parallel languages data and control models are more complicated. The analog in parallel languages is the shared memory model in which all memory locations reside in a single address space and are accessible to all the processors that communicate by reading and writing shared locations. A more decoupled data model is provided by the distributed memory model in which each processor has its own address space of memory locations inaccessible to other processors that communicate by sending and receiving messages.

In parallel languages there are three main control models that determines how processors are coordinated. First, lock-step (vector) synchronization is the simplest model which also known as Single Instruction Multiple Data (SIMD). In this model each processor is either turned off or is required to perform the same operation as all other processors while the active processors at each step work on different data items. Second, Bulk synchronization which is more decoupled control model. In this model the processors synchronize occasionally by using a barrier instruction which means that there is no processor that is allowed to execute a statement past a barrier until all processors have arrived at that barrier which make the processors autonomous and may execute different operations. Third and most decoupled model which known as MIMD (Multiple Instruction Multiple Data) parallelism. In this model two or more processors can synchronize on their own. Synchronizing only for exclusive access to resources or for coordinating access to data that is being produced and consumed concurrently by different tasks.

This paper intends to make a fair comparison among three well-known parallel languages: OpenMP, CUDA, GO to distinguish the main differences and similarities how those languages are handling the synchronization concept.

Figure 1 A Classification of Parallel Programming Languages
2. Problem Statement (Parallel Programming Languages)
In this section, we present a set of several parallel programming languages and describe how they implement the concepts introduced in the previous section. Given a large number of parallel languages that exist, we focus primarily on languages that are in current use and popular, and that support simple high-level task-oriented parallel abstractions.

2.1 OpenMP
OpenMP build out of the need to standardize the directive languages of a few merchants in the 1990s. It was organized around parallel loops and intended to handle compact numerical applications. In any case, as the advancement of parallel programming has developed and the many-sided quality of their applications has expanded, the requirement for a less organized approach to express parallelism with OpenMP has developed.

Synchronization is utilized to force request constraints and to secure access to shared data. There are two levels of synchronization relevant to OpenMP. High level synchronization includes High level compiler directive instructions while Low level synchronization is related to the thread-level locking mechanism.

High Level Synchronization

a. Critical
The critical directive uses to specify a region of code which should execute only one thread at a time. If in the current time a thread is executing inside a critical region and another thread try to reach that region and attempts to perform its command, the last thread will be blocked until the first thread exits that critical region. Furthermore, the optional name permits various and different critical regions to exist the operation, these names perform as global identifiers while different critical regions with the same name are treated as the same region. On the other hand, all critical sections which are unnamed, are treated as the same region. In critical directive, it is illegal to branch into or out of a critical block. In this example, all threads in the team will challenge to execute in parallel, yet, because of the critical construct surrounding the increment of x, only one thread will be able to read/increment/write x at any time.

```c
#include <omp.h>
main()
{
    int x;
    x = 0;
    #pragma omp parallel shared(x)
    {
        #pragma omp critical
        x = x + 1;
    } /* end of parallel section */
}
```

b. Atomic
The OpenMP critical section is extremely broad, and there is an overhead every time a thread enters and exits the critical section. Atomic delivers similar mutual exclusion as the above critical pragma, but it only applies to the update of a memory location. An atomic operation has much lower overhead, It depend on the hardware to do the atomic operation.
c. Barrier
Barrier is one of the important concept when it comes to synchronization. Basically Barrier provides a point till which all the threads has to wait, once all the threads reach that waiting point then the remaining process will be continued further in the same parallel manner and the process will be repeated. The barriers are classified into two types, namely implicit and explicit. The implicit barriers are generally placed at the end of the block which can be ignored by “nowait” clause placed at the end of pragma directive. While “barrier” clause can be used to place explicit barrier following an if, while, do, switch or label along with a point where no or all threads should wait.

```c
#pragma omp parallel
{
    int id = omp_get_thread_num();
    A[id] = big_calc1(id);
    #pragma omp barrier
    #pragma omp for
    for (i=0; i < N; i++)
    {
        C[i] = big_calc3(i,A);
    } // implicit barrier
    #pragma omp for nowait
    for (i=0; i < N; i++)
    {
        A[id] = big_calc4(id);
    } // no implicit barrier due to nowait
} // implicit barrier at the end of parallel region
```

d. Ordered
Ordered is used for sequential execution (specified using “order” region) untill which threads will be executed concurrently and after the “order” region they will be executed sequentially in the same manner as serial loop.

```c
#pragma omp parallel for ordered schedule(dynamic, 4)
for (i=0; i <N; i++)
{
    tmp = foo(i);
    #pragma ordered
    res += consume(tmp);
}
**Low Level Synchronization**

**a. Flush**
Also known as flush set in memory it identifies a point at which the compiler has to ensure that all the threads are viewing a specific objects in the same view. All the visible variables will be the part of the flush set if any arguments are not specified in the flush clause. Flush guarantees that all the operations (if they overlap) like read and write will occur before the execution of flush is completed. All the operations will be paused until the flush gets executed. Also if there are Flushes with overlapping flush, then they cannot be re ordered. Flush will force the data to be updated in memory so that every time the thread will used most recent values. Generally the flushes are active at the entry to and exit from parallel barriers, parallel work sharing (if nowait is not there), ordered and critical region.

```c
#pragma omp sections
{ 
#pragma omp section
  // more about section later
  {fill_rand(N, A);// producer
    #pragma omp flush
    flag = 1;
    #pragma omp flush (flag)
  }
#pragma omp section
{#pragma omp flush (flag)
  while (flag != 1)
    {#pragma omp flush (flag)}
  #pragma omp flush
  sum = Sum_array(N, A); // consumer }
}
```

**b. Locks**
Locks are used to protect the resources, i.e. they imply a memory fence to all the variables of thread. If any variable is set or unset, a nested lock is available to the variable if it is owned by a thread which is currently been executed.

```c
omp_lock_t lck;
omp_init_lock(&lck); 
#pragma omp parallel private(tmp, id)
// init lock
{
  id = omp_get_thread_num();
  tmp = do_lots_of_work(id);
  omp_set_lock(&lck); // wait here for your turn
  printf("%d %d", id, tmp);
  omp_unset_lock(&lck); // release the lock
}
omp_destroy_lock(&lck); // free up storage
```
2.2 CUDA
CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) model created by NVIDIA (wikipedia.org). It allows software developers to use a CUDA enabled Graphics Processing Unit (GPU) for general purpose processing – an approach known as GPGPU. The CUDA platform can be accessed in a variety of different languages through CUDA-accelerated libraries. However, this paper focuses on C++ API on CUDA platform. CUDA offers a variety of synchronization at different levels. Block Synchronization and Stream Synchronization [9].

a. Block Synchronization:
Requires all the threads (fundamental means of parallel execution) within a block to be synchronized; while all threads in a grid execute the same kernel function. Threads within a block usually don't complete their tasks simultaneously. [9] To ensure all threads are synchronized at a certain point in an application synchronization barrier, __syncthreads() is implemented. Once one thread reaches the __syncthreads() call, it will wait until all threads have reached it before executing the next instruction. It can be used to ensure memory transfers are completed, all threads reach the end of the loop, or all threads complete their assignments.

In the following example, a synchronization barrier is placed after the addition operation to ensure all threads complete the addition operation before continuing.

```c
__global__ void addition (parameters)
{
    //some code
    Result[i] = a[i] + b[i];
    __syncthreads();
}
```

b. Stream Synchronization:
A stream is a sequence of commands (possibly issued by different host threads) that execute in order. Different streams, on the other hand, may execute their commands out of order with respect to one another or concurrently. To ensure all streams are synchronized, a synchronization barrier is used. There are two types of stream synchronization[9]:

1. Explicit
   - Placing the synchronization barrier explicitly, to synchronize tasks such as memory operations.
   - All streams created on the device are non-blocking; that is, they do not support implicit synchronization with the default NULL stream. Invoking multiple concurrent kernels requires slightly more programming than invoking one kernel at a time[9].
   - Synchronize everything
     • cudaDeviceSynchronize() : Waits until all preceding commands in all streams of all host threads have completed i.e. blocks host until all issued CUDA calls are complete
   - Synchronize w.r.t. a specific stream
     • cudaStreamSynchronize(streamid) : Takes a stream as a parameter and waits until all preceding commands in the given stream have completed. It can be used to synchronize the host with a specific stream, allowing other streams to continue executing on the device.
   - Synchronize using Events
     1. Create specific 'Events', within streams, to use for synchronization.
     2. cudaEventRecord(event, streamid) : Takes in stream and record the parameter event. Resources associated with event is locked till devices that uses it are completed.
3. cudaEventSynchronize(event) : similar to cudaStreamSynchronize, it waits for devices that uses event that was recorded to be complete [9].

4. cudaStreamWaitEvent(stream, event) : takes a stream and an event as parameters and makes all the commands added to the given stream after the call to cudaStreamWaitEvent() delay their execution until the given event has completed. The stream can be 0, in which case all the commands added to any stream after the call to cudaStreamWaitEvent() wait on the event.

5. cudaStreamQuery() : provides applications with a way to know if all preceding commands in a stream have completed [9].

```{   
  cudaEvent_t event;
  cudaEventCreate(&event); // create event
// host to device copy of input
  cudaMemcpyAsync (d_in, in, size, H2D, stream1);
  cudaEventRecord (event, stream1); // record event
// device to host copy of previous result
  cudaMemcpyAsync (out, d_out, size, D2H, stream2);
  cudaStreamWaitEvent (stream2, event); // wait for event in stream1
  kernel<<<, , , stream2>>>(d_in, d_out); // must wait for 1 and 2
  asynchronousCPUMethod( ... );
}
```

2. Implicit:
- These operations implicitly synchronize all other CUDA operations [9].
  2. Device memory allocation cudaMalloc.
  3. Non-Async version of memory operations cudaMemcpy* cudaMemset*.

2.3 GO
This paper will mainly deal with one of the aspects of GO programming language which is parallelism through synchronization. We will venture into the main objective of creating this language along with fundamental details about this language. Constructs of GO are designed and developed by Google to solve their own problems. Google took initiative to fulfill demands of ongoing concurrent programming and degree of data which a company handle everyday by creating GO. Robert Griesemer, Rob Pike, and Ken Thompson are the creators of this language but now GO is an open source language. Rob pike have mentioned in his introductory speeches of GO that this language is born out of their frustration of building software to handle concurrency. Go has very interesting concepts and ideas in object oriented paradigm other than concurrency. There is a challenge of multi core programming in GO which will make parallelism easy. An in depth analysis of parallel programming in go along with multi core parallel go will be discussed in the next sections.

Synchronization in GO language
In GO language, there are some packages which provides synchronization. A package called ‘sync’ in GO provides basic synchronization through mutex (mutual exclusion locks). Channels and communication are used to achieve higher level synchronization.
Concurrency and Channels in Go
Concurrency described as the composition of independently executing processes, on the other hand parallelism is the simultaneous execution of (possibly related) computations. Concurrency is about dealing with lots of things at once where Parallelism is about doing lots of things at once. Concurrency is about structure of the task being executed while parallelism is about the how is the task executed. [1] Channels in Go language is to serve synchronized execution for concurrent processes. Also it can be used to send and receive values with channel operators ‘<-‘.

Channel Synchronization in GO example- [2]

```go
package main
import "fmt"
import "time"

func worker(done chan bool)
{
    fmt.Println("working...")
    time.Sleep(time.Second)
    fmt.Println("done")
    done <- true
}

func main()
{
    done := make(chan bool, 1)
    go worker(done)
    <-done
}
```

Goroutines in GO
In GO language, concurrency can be achieved using Goroutines. Goroutine is nothing but a light weight process. It executes with other Goroutine concurrently inside the same address space. We can achieve light weight and efficient functions which can complete task with better efficiency. Our main goal in this section is to understand Goroutines and parallelism together. The following section compares Goroutines with coroutines.

Goroutines vs. Coroutines
There are two keywords yield and return in coroutines to suspend and resume the execution of a program. It also controls the flow of an execution. This seems more like preemptive where a process can be suspended whenever it is needed. On the other side in Goroutines the control is transferred from one point to other point which seems like a non-preemptive execution. The control transferring process is hidden from a programmer which is an inherent functionality of a GO language.

Goroutines have many interesting features out of which is that multiple Goroutines can stay in a single thread and switch in between operating system threads. This concept resembles the concept called as green thread in computer programming which emulates the multithread environment and work on different operating systems. During comparison of the different scenarios in between Goroutines and coroutines, I have reached to this point that a preemptive scheduler yields better results and is more capable of executing complex tasks which needs only preemptive execution. Go uses concept of channels and threads to gain concurrent and efficient results. Coroutines does not have parallel or concurrent execution. They just provide an illusion for parallelism.
Goroutines and Parallelism
Goroutines are used in Go to create a function or a method. Goroutines are lightweight because of their less memory and resource usage. Also initial stack size of Goroutines is small. In general operating system schedules threads to run against. For execution of all Goroutines, the Go runtime assigns a single logical processor. To get better performance and efficiency all Goroutines can be scheduled to run concurrently with single logical processor and OS thread. Though it is not suggested to add more than one processor, GO language gives an ability for it using the GOMAXPROCS runtime function. GOMAXPROCS [3] variable limits the number of operating system threads that can execute user-level Go code simultaneously. There won’t be any limit on the threads which gets blocked in a system calls from GO code. That does not comes under GOMAXPROCS limit.

This package's GOMAXPROCS function queries and changes the limit. GOMAXPROCS can be set explicitly using the GOMAXPROCS environment variable or by calling runtime. GOMAXPROCS from within a program. For example, if GOMAXPROCS is 4, then the program will only execute code on 4 operating system threads at once, even if there are 1000 Goroutines. The limit does not count threads blocked in system calls such as I/O. [4]

With Go 1.5 release GOMAXPROCS is set by default to the number of CPUs available. This change has significant implications of the notion of parallelism in GO language. Given this change performance of single-goroutine programs can improve by raising GOMAXPROCS due to parallelism of the runtime, especially the garbage collector and the performance of multi-goroutine programs with real parallelism can scale linearly with GOMAXPROCS.

GOMAXPROCS example-

```go
Func Parallelism() int {
    maxProc := runtime.GOMAXPROCS(0)
    numCPU := runtime.NumCPU()
    if maxProc < numCPU {
        return maxProc
    }
    return numCPU
}
```

3. Proposed Approach and Outcomes
OpenMp, CUDA and GO are all parallel programming languages uses MIMD structure. Each of them are inspired by different reason and uses different concepts to aid their effort in parallel processing. OpenMp and GO uses the CPU while CUDA uses the GPU or sometimes called GPGU to implement parallel processing. OpenMp processes in thread level. GO mainly processes in Goroutines and channel. CUDA processes in thread, device, event and stream level.

- Mutex (Mutual Exclusion Lock)
GO uses type called Mutex(mutex.go), while OpenMp uses Critical, Atomic and Lock but CUDA does not have an intuitive way of achieving mutex. In Critical (#pragma omp critical (dataupdate)), threads try to access critical region of other threads, will be blocked. Atomic (#pragma omp atomic) uses Critical but block memory location instead of code. Furthermore, OpenMp has lock called omp_lock_t that locks threads based on programmer’s preference. CUDA uses a function called atomicCAS and assign mutex as a value and compare it with other threads or uses guards to ensure threads does not overlaps.
- **Barrier**
OpenMP has Barrier (pragma omp parallel) that could be implemented implicitly and explicitly. Explicit way will be to use nowait (prama omp nowait). CUDA has barrier namely: syncthreads which wait for all the threads in a block to be done before advancing front. However, since it also uses both device and stream level, thus barrier uses for them are cudaDeviceSynchronization, cudaStreamSynchronization and cudaEventSynchronize. GO uses a type called WaitGroup to wait for a group of goroutine to be completed.

- **Semaphores**
Both OpenMP and CUDA does not have an intuitive way of achieving Semaphores. OpenMP has to use Lock (omp_lock_t) while, CUDA similarly to mutex, uses AtomicCase to achieve Semaphores. In GO, can achieve semaphores by having a buffered channel.

<table>
<thead>
<tr>
<th></th>
<th>OpenMP</th>
<th>CUDA</th>
<th>GO</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GPU</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Goroutine</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>OpenMP</th>
<th>CUDA</th>
<th>GO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutex</td>
<td>✓ (Critical, Atomic)</td>
<td></td>
<td>✓ (Mutex type)</td>
</tr>
<tr>
<td>Barrier</td>
<td>✓ (Barrier)</td>
<td>✓ (synchthread,…)</td>
<td>✓ (Barrier type)</td>
</tr>
<tr>
<td>Semaphores</td>
<td></td>
<td></td>
<td>✓ (Buffered Channel)</td>
</tr>
</tbody>
</table>

4. Conclusion
This paper presents, an overview of synchronization in three parallel programming languages: OpenMP, CUDA and GO. These languages are in current use, popular, offer rich and highly abstract functionalities. The paper also introduced how parallel computation is different from serial computation, three main control models in parallel programming model: SIMD, Bulk synchronization, MIMD. Types of High level synchronization and Low level synchronization in OpenMP. How block synchronization and stream synchronization is handled in CUDA and packages that manage synchronization in GO; also how Goroutines are different from coroutines in Go. In addition the paper also provides a comparative evaluation of how synchronization is handled in all these three languages.
This paper shows how the concept of collective and point-to-point synchronization in programming a parallel system works and are managed in the languages OpenMP, CUDA and GO. In the parallel languages the shared memory model in which all memory locations are present in a single address space and are accessible to all the processors and they communicate by reading and writing the shared memory locations.

Starting with OpenMP, which circles around parallel loops and is intended for the compact numerical application in which synchronization is used to force the request constraints and to secure the access to shared data. Depending upon the mechanism the synchronization in the OpenMP is divided into two levels namely High level (for compiler directive instructions) and low level (for thread related locking mechanism). Here for the high level synchronization, the directives like Critical, Atomic, Barrier and order are discussed and for the low level synchronization directives like flush and locks are discussed. CUDA (Compute Unified Device Architecture) is a parallel computing platform and API model created by NVIDIA. Here in this paper we are focusing on C++ API of CUDA platform. Here the synchronization is divided into two parts namely block (threads in one particular blocks will be synchronized) and stream (stream of threads are synchronized). Here stream synchronization is further divide into two i.e. explicit (for streams created on non-blocking device) and implicit (for self-synchronizing CUDA operations).

In GO there are some packages namely ‘sync’ that provides synchronization using the mutex i.e. mutual exclusion, also use of channel and communication makes the higher level of synchronization possible. Further the concept of Goroutines is described in the paper. Basically the control transferring process in coroutines is performed through suspend and resume, but in the goroutine it is hidden from programmer which is like a non-preemptive execution.

Throughout the paper thread synchronization was the main goal. Multiple thread should not execute the critical part of program segment at a time, if one thread is executing that part, then other threads should wait until the first thread finishes. If proper synchronization is not implemented then it may lead to unpredictable values of variables and will result to false answers. Even though there are different methods and ways to implement this concept in different language, the end result is always going to be the same. And as we mentioned during the presentation all the languages i.e. OpenMP, CUDA and GO and great in their own context and the requirement of the system and the choice of the language will lead us to the use of the above mentioned concepts.
References

[10] https://cvw.cac.cornell.edu/gpu/synchronization

Mutex
(GO’s mutex: https://gobyexample.com/mutexes)
(OpenMP’s mutex : http://bisqwit.iki.fi/story/howto/openmp/)
(CUDA’s mutex: https://devtalk.nvidia.com/default/topic/416886/mutex-for-cuda/)
(CUDA ‘s mutex: http://cs.umw.edu/~finlayson/class/fall14/cpsc425/notes/22-cuda-sync.html)

Barrier
(Go’s barrier: https://golang.org/pkg/sync/#WaitGroup)

Semaphores
(Open MP’s semaphores: http://hpc.iucaa.in/Documentation/hpc_training/tcs/HPCTrainingmcore.pdf)
(GO’s Semaphores: http://www.golangpatterns.info/concurrency/semaphores)