3. Read the paper by Castelló et al. that considers lightweight threads versus sticking with operating system threads. Then answer the following questions:

a) Discuss some of the tradeoffs of using OpenMP alone versus adding the capabilities of light-weight threading libraries described in the paper.

According to the paper, OpenMP is better suited to parallel computing on a single computer shared memory architecture. Due to the use of shared memory between threads to coordinate parallel computation, it is highly efficient on multi-core/multi-CPU architectures, has a low memory overhead, and has simple and intuitive programming statements, making it easy to program and easy for compilers to implement. However, OpenMP suffers from performance degradation when working on massive or multi-array parallel environments. In this case, we could use lightweight threads techniques as tradeoffs, which provide more efficient context switching and synchronization mechanisms to somehow improve the performance in massive parallellelism.

b) Select one of the lightweight threading libraries discussed and provide an example of how you would use it to parallelize a simple vector addition kernel (adding two vectors of single-precision floating point numbers together). You do not need to provide code that compiles, just provide enough syntax to show you know how to use the library.

I choose Golang. Unlike traditional multi-threaded communication through shared memory, CSP of Golang is about "sharing memory by communication". Golang's CSP concurrency model is implemented through goroutines and channels. A goroutine is the unit of execution for concurrency in Golang and it is actually similar to the traditional concept of a "thread". A channel is a communication mechanism between concurrent constructs (goroutines).

Generally speaking, it is like a "pipe" for communication between goroutines, somewhat similar to the Linux pipe. Golang uses a two-level thread model. This thread model creates multiple kernel-level threads and then uses its own user-level threads to correspond to the multiple kernel-level threads created, with its own user-level threads being scheduled by its own program and the kernel-level threads being scheduled by the operating system kernel.

The Golang mechanism for parallel summation of arrays is as follows: the main

function imports the summation of each array segment through a loop, waits for all goroutines to finish execution, and obtains the final value sum.

```
sum = 0
start = START
for i = 1; i <= STEP; i++ {
     go childSum(start, LEN)
     start += LEN
}
for runtime.NumGoroutine() > 1 {
      time.Sleep(100 * time.Millisecond)
}
fmt.Printf("%d\n", sum)
```

childSum is the main function that the goroutine runs. This function first computes the sum of the sub-arrays by looping over them, and then accumulates the sum of the sub-arrays to the sum variable, where sum is guaranteed to be correct using a mutex.

c) The paper evaluates the performance of BLAS-1 functions. Discuss what is included in the BLAS-1 subprograms.

BLAS-1 functions are fully compatible with LWT fine-grained approach and are highly parallelizable. Sscal multiplying a component of a vector by (and overwriting) a scalar. To achieve task-level parallelism in these subprograms, the elements in the vector are divided into threads and this granularity of parallelism does not hide the overhead of thread management.