Silk - C Library for fork-join style parallelism

Ayush Agrawal
ayushagr@andrew.cmu.edu

Abstract— Fork-join parallelism is one of the ways of describing parallelism where execution branches off in parallel at designated points in the program, to join (merge) at a subsequent point and resume sequential execution[1]. It is particularly useful when it comes to parallelizing recursive algorithms. This paper describes the implementation of Silk - a library written in C that provides the users with an interface to parallelize their recursive algorithms using the fork-join model. The library is inspired from the “cilk” library implemented at MIT[2].

Keywords— fork-join, parallelism, recursion, C, distributed work queue, random victim selection, dynamic scheduling

I. BACKGROUND
Parallel libraries/frameworks come in all flavors. Some are tailored specific to a domain while others are meant to be more general purpose. They differ significantly in their ease of use and the amount of control they provide to the user as well. A general workflow of specifying parallelism to mapping it onto the hardware consists of the following four steps:

1. Decomposition
2. Assignment
3. Orchestration
4. Mapping

The meaning of the four steps are explained in this section along with a brief description of what silk does in regard with each of them. Detailed explanation of how silk each of them can be found in Section II of this paper.

Decomposition refers to how you break down a problem into smaller subproblems that can be dealt with in parallel. The subproblems can have dependencies on each other and they need not all be of the same size. Since Silk is meant to be a general purpose library for parallelizing recursive algorithms, it leaves the decomposition of the problem to the user of the library.

Assignment refers to assigning the subproblems created during decomposition to workers. A worker is an execution context which can be a process, thread, task, event, etc. Assignment of work can be left to the user of the library or can be conducted by the library itself or can be a mix of both. For Silk, a worker is a POSIX thread[4]. I choose to use posix threads for their compatibility and wide support. Note that many platforms offer certain extensions to the POSIX threads API. Silk refrains from using any such extension. Silk performs assignment of work on its own. It uses dynamic scheduling to do so[3].

Orchestration refers to the necessary communication between the workers to ensure correctness and efficiency. Workers often need to talk to each other to signal things like availability of work, state of shared queues, end of a phase of computation, etc. The current version of Silk uses the implementation of mutex provided by the pthread library to implement synchronization among workers. Any another form of orchestration required to implement an algorithm is left for the user of the library to implement and is not part of the library itself.

Mapping refers to mapping the workers to actual hardware units. Most libraries won’t interface with the hardware directly and instead interface with the operating system and provide a mapping between worker and an operating system construct. Since silk uses POSIX threads, it simply uses embodies the mapping used by the pthread library which at the time of writing this paper meant a 1:1 mapping between user level threads and kernel threads.

Since there is an overhead associated with managing the workers, during the decomposition phase, the user of the library may wish to stop subdividing a problem when the size of the problem becomes less than a certain threshold and start using a sequential algorithm for problems of that size. This is necessary from the point of efficiency and is not a requirement by the library. The choice of the threshold is left to the user as well.

II. IMPLEMENTATION

A. The Silk Interface
The Silk library exposes the following functions to the user of the library:
Part C.

The silk_init function is used to initialize the Silk library. It takes as an argument the maximum number of threads that can be spawned by the Silk library including the thread that initializes the Silk runtime. Specifying a value of zero or less will initialize the Silk library with a default value of 8 threads. This function should be called before any other routine from the Silk library is used.

The silk_release function is used to release the resources held by the Silk library. This function should only be called once the user no longer wishes to use the Silk library. silk_init can be issued following silk_release to reuse the library.

The silk_fork is the function used to specify “potential parallelism”. Since the library is dealing with recursive algorithms, a unit of work is described using a function and the argument of the function. The prototype mandated for the function is:

```c
void (*silk_work_routine)(void*, int)
```

The first argument to the function is the same as what is passed to the silk_fork routine and the second argument to the function is the worker id of the worker executing the function. The worker ids are assigned by the silk framework. Their use is described in Part C of this section. The user of the application is responsible for the memory management of the first argument supplied to the function.

Lastly the silk_join routine is used by the “master” thread (the first thread to call fork_join) to reap all the other threads using pthread_join. Placement of the silk_join routine is very crucial. If one chooses to place the silk_join routine in the recursive function itself then the argument passed to the silk_work_routine can be allocated on the stack, otherwise it must be allocated on the heap to avoid corrupting the stack. The former might seem more performant but severely limits the size of the problem that can be handled before the stack size imposed by the operating system gets exceeded. More information on this in Section III Part C.

B. The Silk Versions

The Silk library is available to use in four different versions:

1. CoFSQ
2. CoFDQ
3. ChFSQ
4. ChFDQ

“CoF” refers to the continuation first behavior and “ChF” refers to the child first behavior. Continuation first means that the thread calling silk_fork will add work passed to silk_fork to the work queue and then continue on. Child first means that the thread calling silk_fork will actually perform the work specified to silk_fork and add it’s continuation as work in the work queue. Implementing “CoF” doesn’t require any extra effort because it matches the control flow naturally implemented by a compiler. On the other hand, implementing “ChF” requires storing the continuation state of the current thread and allowing other threads to restore to that state. One way of achieving this is using the setjmp and longjmp routines provided by the C runtime. However there is a catch with using setjmp and longjmp. To avoid stack corruption, the routine where setjmp gets called should never return to the caller[5]. This would severely limit the way silk_fork and silk_join gets implemented. Therefore Silk expands the definition of the silk_fork routine in it’s “ChF” versions and takes as an argument both the child and the continuation:

```c
silk_fork(child_func, child_args,
continuation_func, continuation_args, id)
```

“SQ” refers to the single work queue versions where there is a global work queue every worker thread is operating on. “DQ” refers to the distributed work queue versions where each worker thread has a work queue of its own and it even steals work from the work queue of other workers when it has no work left in it’s own queue.

Section III of this paper will describe the results obtained from the running these four versions for the problem of quicksort. One can interpret the result to see that “in general”, the distributed work queue version is more performant and the child first version leads to smaller queue length. However the library ships in all four flavors for the users to decide which version they want to
use. This enables comparison of the four versions across various problem sets.

C. Data Structures

The work queues are implemented using a double linked list and the list of workers is implemented using an array of fixed size. The size of the array is known since the user of the silk library specifies the maximum no of threads to swapn in the call to silk_init.

Apart from these two data structures, global state is maintained to know the number of currently active threads and to hold a handle to the master thread. A handle to the master thread is necessary since only the master thread reaps the worker threads, i.e., worker threads don’t reap any other thread and therefore it is necessary to distinguish the master thread from the other threads in the silk_join routine.

The library very cleverly avoids using a hashmap to map pthread ids to silk ids. For example, in the silk_fork routine in case of a distributed work queue, the worker threads need to know which queue to push the work into. One way to do it would have been to use pthread_self inside the silk_fork routine to get the pthread assigned id to the worker and then use a hashmap to map it to the structure maintained by the silk library to hold information about a worker. Instead the silk library passes to the silk_work_routine, described in Part B of this section, the silk id of the worker and takes it in as the last parameter to silk_fork. Thus need for a hash map is avoided. However this requires the user to specify the silk id when it calls the recursive routine. The only safe value that can be used by the user is the silk id of the master thread (which is 0 for the current version of Silk). Specifying any other value may lead of unspecified behavior.

D. Join

The implementation of silk_join is not what one would usually expect from a join routine like pthread_join. Logically a join routine marks a point in the execution of the program following which all the work issued before the join is guaranteed to have been successfully completed and the resources created before the join are guaranteed to be reaped (released). Silk follows the same semantics for the master thread but for all other threads silk_join is a no-op. This follows from the fact that Silk treats all the threads it swapns as a single pool of workers and individual threads don’t keep track of the threads they spawn.

E. Dynamic Scheduling

Works of different sizes can be placed in the work queue(s). Once a thread completes the work at hand, it goes and fetches another unit of work from the work queue(s). In case of distributed work queues, if a thread can’t find any work remaining in it’s own queue, it randomly selects a victim (the thread to steal the work from) and tries to get work from the victim’s work queue. The current implementation is not very efficient and relies on a certain number of tries before giving up and terminating (hardcoded to 25 at the time of writing this paper). A better approach would be to fetch more work from threads having a lot of work in their work queue and being able to avoid empty work queues since they simply waste a try.

III. Results

To verify the correctness of the library and gather statistics on it’s efficiency, I have implemented three recursive algorithms using all the four versions described in Section II Part B. The algorithms implemented are: fibonacci, quicksort and matrix-vector multiplication. This section presents the results obtained from implementing quicksort using Silk.

The specification of the machine used to obtain the results is as follows:

OS: macOS Mojave Version 10.14.6
Processor: 2.4 Ghz Intel core i5
Memory: 8 GB 2133 MHz LPDDR3
No of processors: 1
No of cores: 4
L2 Cache (per core): 256 KB
L3 Cache: 6 MB

The pseudo code for the sequential implementation of quick sort is as follows:

```c
seq_sort(arr, len)
seq_quick_sort(arr, 0, len - 1)
seq_quick_sort(arr, low, high):
    if high <= low + RECURSION_CUTOFF:
```
insertion_sort(arr, low, high)
else:
    j = partition(arr, low, high)
    seq_quick_sort(arr, low, j - 1)
    seq_quick_sort(arr, j + 1, high)

The implementation assumes the existence of two helper routines: insertion_sort that sorts the array for the given range using the insertion sort algorithm and partition that finds the pivot used in the quick sort algorithm.

The pseudo code for a possible implementation (and the one used to obtain the results) of quick sort using Silk is as follows:

parallel_sort(arr, len)
    args = create_qsort_args(arr, 0, len - 1)
    parallel_quick_sort(args, 0)
    silk_join();
parallel_quick_sort(args, silk_id):
    arr, low, high = args.arr, args.low, args.high
    if high <= low + RECURSION_CUTOFF:
        insertion_sort(arr, low, high)
        return
    j = partition(arr, low, high)
    args_left = create_qsort_args(arr, low, j - 1)
    args_right = create_qsort_args(arr, j + 1, high)
    free(args)
    silk_fork(parallel_quick_sort, args_left, silk_id)
    parallel_quick_sort(args_right, silk_id)

The implementation assumes the existence of another helper routine called create_qsort_args which creates the argument for the parallel_quick_sort routine on the heap. Note that the argument is freed once it’s no longer required.

A. Runtime/Performance evaluations

The following graph describes the results of running the quicksort algorithm using the CoFSQ version of the Silk library.

The graphs plots time taken to perform quick sort on the y-axis and the implementation used on the x-axis. The different colored lines represent different input sizes.

As one can see, the implementation written using Silk is better than the sequential implementation (other than the case where Silk is invoked with just a single thread due to the overhead invoked in managing the thread). The performance gain is more for large input sizes because there is more opportunity to amortize the overhead cost of managing the worker threads. For small input sizes overhead of managing the threads dominates. But the performance gain is not linear and as we start to increase the no of threads over a certain point, either the performance gain keeps on getting smaller or even sometimes we see degradation in performance. This is due to the fact that there is just a single work queue and multiple producers and consumers are contending for a lock over it. As the number of threads keep on increasing the queue becomes the bottleneck.

The following graph describes the results of running the quicksort algorithm using the CoFDQ version of the Silk library.
The graphs plots time taken to perform quick sort on the y-axis and the implementation used on the x-axis. The different colored lines represent different input sizes.

If we compare the time evaluation output of CoFSQ and CoFDQ, we can see CoFDQ performs better in general. It is worse in the case of smaller no of threads (2 thread for the graphs shown) due to continuous work stealing from the limited no of queues. Once the no of threads becomes sufficiently large enough (4, 8 threads for the graphs shown) we begin to see the benefits of using distributed work queues. This is due to the distributed work queue. There is no more a single queue which is becoming the point of contention. However for very large number of threads (16 for the graphs shown), the performance degrades. This has more to do with the platform the tests were run. Since the machine only has 4 cores, running 16 threads on it none of which does IO overwhelms the advantage obtained from parallelism. The distributed work queue was implemented using a random selection scheme for the victim when a thread doesn’t find any work left in its personal work queue. I suspect it to be the cause of bottleneck in some cases since my implementation is not very mature and often checks for work items in queues which are empty. The first graph in the Appendix gives a clearer performance comparison of CoFSQ and CoFDQ.

B. Max queue length evaluation

The following graph describes the results of running the quicksort algorithm using CoFSQ version capturing the maximum size of the work queue observed.

If we compare the max queue length from the CoFSQ version with the ChFSQ version, we see that the max queue length is lesser for the child first approach. It is in agreement with what was observed with cilk[2]. The reasoning behind this is that with the child first approach there is less time for work items to get accumulated in the work queue. Note as described in Section II, the child first implementation of Silk doesn’t use state store and restore. Due to this there isn’t work involved in doing the setup for continuing with the work fetched from the work queue (like state restoration for a thread, etc). Therefore the difference observed is not as much. This may be more performant but implementation of
the child first policy in Silk is limited to few use cases only and cannot be used in a general purpose manner like the one implemented by “cilk”. The second graph in the Appendix gives a clearer comparison between CoFSQ and ChFSQ.

C. silk_join placement

As mentioned in Section II, the placement of silk_join can determine whether it is necessary to allocate the arguments passed to the parallel recursive routine on heap or not.

To check the limits, the quicksort implementation using Silk was modified giving rise to three different versions. Note the operating system where the results were obtained imposed a limit of 8 MB on the stack size and that the silk library was invoked with no of threads equal to 1 to get the worst case estimate.

In version 1, silk_join was called right after specifying the recursive calls, i.e., it was part of the recursive function itself. It enabled us to declare the arguments being passed to the parallel quicksort routine on stack itself but at the same time meant deep nested recursion for the master thread in the silk_join routine. The maximum size of the input that was tolerable was 2198600 (about 2 million entries).

In version 2, the placement of silk_join wasn’t changed but the arguments were allocated on the heap. The maximum size tolerable jumped up to about 4.3 million entries.

In version 3, the silk_join routine was placed outside the recursive call. The maximum size tolerable became more than 500 million entries.

REFERENCES


APPENDIX

The graph above shows a comparison between CoFSQ and CoFDQ. On the x-axis we have the time taken in milliseconds and on the y-axis we have a pair of number of threads, input size.

The following graph shows a comparison between CoFSQ and ChFDQ. On the x-axis we have the observed queue length and on the y-axis we have a pair of number of threads, input size.