1st Project Report

Koukoulis Ippokratis 2324

1. Summary

This report contains two main sections where the design decisions of the implementation are presented and the experimental evaluation of the main optimizations. For more details on the implementation view the source code.

2. Implementation and design decisions

2.1. User threads

User threads are created to be generally equal with uniform behavior (except for the main user thread where exiting is handled differently). At the last-highest 8 bytes of its stack the pointer to its thread_descriptor is stored. Each user thread is assigned to the native thread that "owns" the user thread that created it ,and thus its "ownership" is set to that native thread. The concept of "ownership" is the link between the native thread "the owner" and the user thread that it owns and it is represented by a pointer to the native thread (in the thread descriptor struct) that "executed" (and thus owns) this user thread .This pointer-link is used by the user level thread to access its native thread struct (for example to return to scheduler context when it is terminated to free its resources).Each user level thread is wrapped by a function which handles the "bookkeeping" of termination. User level thread descriptors are statically allocated (when the programmer declares a thread_t variable).

2.2. Native threads

Native threads are stored in a global array that is dynamically allocated at the time of thread_lib_init.The native thread struct consists (mainly) of these fields:

- The context of the scheduler of this native thread. Note that with the exception of the first native thread , all the other native threads that are spawned after thread_lib_init serve as the scheduling context of themselves while for the first native thread we create a different context to serve as its scheduler.
- A pointer to the current running thread that runs on this native thread, which cannot be touched by any other native thread(this thread is detached from the queue).
- A queue of the threads that are ready to run (doubly linked double ended list) which can be accessed by other native threads which want to steal.
- The pthread id of the native thread (necessary to do joins).
- And and idx field which is used to find the position of the thread in the array.

2.3. Scheduling

Initially all of the scheduling logic was executed inside the scheduler context of each native thread. However in some cases this leads to unessecary context switches (mainly when yielding).Thus the logic of scheduling is split across the thread_yield function and the scheduler context. The scheduler function manages threads that exit and

work stealing in case we dont find a thread in our queue. Yield tries to find a thread in the native thread's queue,
if it fails(in case everything was stolen which would be rare)
it switches to the scheduler.This results into slightly better execution times when many yields happen.

2.4. <u>Work stealing</u>

Initially for work stealing when a native thread found that it had no threads on its queue , it searched for threads to steal starting from its right neighboor , doing a round robin until it reached its left neighboor (neighboors meaning neighboring indexes in the native threads ie for 4 threads thread 2 would search in the order 3->0->1).If it found a queue which had threads then it stole a thread from this queue. This generally is inefficient since some native threads gather a lot of work at the beginning of the program and threads that try to work steal end up creating contention between themselves because they constantly try to access native thread queues. To reduce the amount of contention we try to steal more than just one thread and instead we steal half of the threads from the “victim” native thread. More specifically, we steal half of the threads from the end of the victim, and add to them to our own. This lead to better execution times up to 15%. Finally ,a different scheduling approach was also tried which instead chooses the victim queue according to the size of the queue where the queue with the most threads is the one we chose to steal from. Although the size of the queues can be inaccurate from the time we read it until we actually try to steal (we could lock all queues to ensure that the sizes of the queues don’t change but this would lead back again to increased contention), it is sufficient to make an educated guess on which queue is the most appropriate to steal from (to have a better load balancing).This approach yield similar results to the previous method. Both of them are included in the source code and can be enabled-disabled by defines (although the neighboor policy is the default one, both work).

2.5. <u>Termination</u>

When thread_lib_exit is called it first checks if all user threads (except the user thread that called it since it is implicit that this thread is also going to terminate) have terminated.Each user thread decrements a count when it terminates (either by thread_exit or returning).If not we yield and try to execute other threads to help the other native threads to finish faster (this also guarantees that all threads will be executed in the case where we have 1 native thread and we didn’t wait for the spawned user threads to finish).After all the other threads have finished we signal (via a flag) the other native threads to terminate and we join them. After joining we free the resources of the library.
Functionality to call thread_lib_exit from other threads except the main thread is supported by defining the ATEXIT_HANDLER ,which adds an atexit handler to handle the exit of main (where the decrement of the count happens) and we switch to the scheduler.When ATEXIT_HANDLER is defined all the user threads check if the thread count is 0 after dercrementing it when they exit, to terminate the native threads ,since if no user level threads have remained then there is no remaining code to be executed (since main has also terminated) and thus the native threads and by extension the process must terminate. All the tests-benchmarks can be used without the need to define ATEXIT_HANDLER (because the main thread is always the last one to terminate- call thread_lib_exit).There is an example

where another thread is the one which calls thread_lib_exit,for this example the ATEXIT_HANDLER must be defined to work (all the other examples will also work with ATEXIT_HANDLER defined , the reason I chose to have it to be optionally defined is because there is a slight(but mostly negligible) performance penalty when it is defined since every thread has to check the thread count at termination, and because this extension uses undefined behavior to work by doing essentially 2 exits although even in this case we should be fine since we always check the thread count at the atexit_handler and thus it shouldn't be possible for the process to go on a recursive loop. But nonetheless its still undefined behavior. Also if ATEXIT_HANDLER is defined we cannot call exit() explicitly from other user threads to terminate the process, this would completely break the library.)

2.6. Semaphores

Semaphores are statically allocated and implemented with a spin_lock and a queue to keep blocked threads in FIFO order.The semaphore can take positive values and it cant go below 0 (similar to the Linux semaphore posix implementation).The semaphores are statically allocated. When a sem_up happens and there are blocked threads in the queue we take the first thread of the queue and we "wake" it up by moving it at the begging of our ready queue (without increasing the value of the semaphore).Thus we don't create unnecessary contention for the semaphores.

3. Experimental evaluation

For all the benchmarks that were used , the total execution time was measured with the shell time command (using the real time measured).All benchmarks were mainly tested with 4 native threads (more threads were also tested and they work, but since my processor has 4 physical cores the max degree of parallelism is 4 and the maximum speedup is gained by 4 native threads).The experiments were conducted on an Intel i5-6500 CPU on an Ubuntu 20.04 Virtual Machine with 8 GBs of RAM allocated to the VM.A mean of executions is used for all experiments.

3.1. Stack recycling

In this implementation stack recycling is employed to make thread creation cheaper and use stacks from other threads that were terminated. The create_bench is used to measure this gains of stack recycling which creates 20*1024 threads which themselves create another 1024/128 threads (inspired by the multilevel example) that do nothing. This is done at 1024 rounds (we wait for all threads of a round to terminate to proceed to the next).Setting the MAX_SIZE limit to 0 (which means that we don't keep any stacks and essentially disabled the optimization), the mean real execution time (measured from 10 consecutive executions) is 6.61 seconds. Setting the MAX_SIZE stack reservoir limit to 1000 the mean real execution time is 1.95 seconds ,having a 3.38x speedup compared to the unoptimized execution where we don't recycle stacks.

3.2. Work stealing optimizations

Work stealing has been optimized to steal half of the threads from a queue , than just stealing one thread to reduce contention of the queues.As

previously said both the NEIGHBOOR_POLICY and MAX_POLICY yield similar results across all benchmarks (although this may not be true if we tried different numbers of threads ,or different benchmarks , but at least for what I tried they were pretty much on equal ground.)
This optimization mainly focuses on the benefit we get from stealing more than just one thread. The matmul_multilevel benchmark was used to measure the speedup compared to the unoptimized neighborhood policy where we steal one thread. The unoptimized policy has a mean real execution time of 3.26 seconds while the optimized policy that steals half threads of a queue has a mean real execution time of 2.79 seconds resulting in a 1.16x speedup. For reference the max policy also achieved 2.79 seconds of execution time.

3.3. <u>Yielding switches</u>

Finally unnecessary context switches when yielding were also optimized so we don’t have to switch to the scheduler and back to another thread.Instead when a thread yields it switches immediately to the next thread in its queue.The matmul_yield benchmarks was used to measure the benefits of this optimization.A previous unoptimized version of the code where we always switch to the scheduler had a real mean execution time of 3.21 seconds while the optimized and final version of the code has a mean execution time of 2.98 seconds resulting in a 1.07x speedup.