CSE113: Parallel Programming

Homework 3: Concurrent Data Structures
Assigned: May 7, 2021
Due: May 21, 2021

Preliminaries

1. This assignment requires the following programs: `make`, `bash`, and `clang++`. This software should all be available on the provided docker.

2. The background for this homework is in the class material for Module 3. The last two lectures should be emphasized.

3. Find the assignment packet at `https://sorensenucsc.github.io/CSE113-2021/homeworks/homework3_packet.zip`. You might collect this from a a bash cli using `wget`. That is, you can run:

   ```
   wget https://sorensenucsc.github.io/CSE113-2021/homeworks/homework3_packet.zip
   ```

   Download the packet and unzip it. But do not change the file structure.

4. This homework contains 2 parts. Each part is worth equal points.

5. For each part, read the *what to submit* section. Add any additional content to the file structure. To submit, you will zip up your modified packet and submit it to canvas. If you have questions about the structure of your zip file, please ask!

6. It is okay to discuss docker questions and file structure questions with your classmates. Please don't share results until the week after the assignment has been posted. Please do not share code at all.

7. You should feel free to modify any part of the function that you are working on. In some cases I have left some loop structure in the code to guide you, but in some cases you may need to change loop bounds, increments, etc.

1 Producer-consumer Queues

In this part of the assignment you will implement several variants of a producer-consumer queue and optimize a simple concurrent program using your queues.

The set-up is as follows: You are programming a strange machine: This machine can execute two threads: a memory thread, which can access memory; and a trig thread, which can execute trigonometry functions. The two threads can communicate with each other through producer-consumer queues.

The overall goal of the program is to take in an array of floating point values and compute the cosine of each value, and then store the value back. To do this, the array of floats is sent to the memory thread. For each item in the array, the memory thread loads the value, and sends it to the trig thread for computation. The trig thread performs the computation, and sends the value back to the memory thread. The memory thread then stores the value back to memory.

There are two producer-consumer queues: `memory_to_trig` has the memory thread as the producer and sends values to the trig thread (the consumer). The `trig_to_memory` queue sends values from the trig thread (the producer) to the memory thread (the consumer).

While this scenario may seemed contrived, it is similar to how some accelerators are programmed.

You have 4 parts of this assignment:

1.1 A synchronous producer-consumer queue

For this part, the program is given in `main.cpp`. Your job is to implement `CQueueSync.h`. The queue should be implemented in a synchronous way, i.e. as discussed in the May 6 lecture. Every equeue must wait for the corresponding dequeue, and vice versa. This program is built with the first compile line of the makefile and produces an executable called `syncQueue`.

1.2 An asynchronous producer-consumer queue

Similar to the above, the program is given in `main.cpp`. Your job is to implement an asynchronous concurrent queue, as discussed in lecture. You will use a circular buffer of size `CQUEUE_SIZE`. You will implement this queue in `CQueue.h`. This program is compiled in the second line of the makefile and will be called `asyncQueue`.

1.3 Optimizing programs to use asynchronous producer-consumer queues

In this part of the homework, you will use your asynchronous queue of the previous question. This time you will modify the client program given in `main3.cpp`. Your job is to optimize the program to more fully take advantage of asynchronous queuing. To do this, the communicating threads should batch their communication. That is, they should communicate 8 items at a time, using 8 consecutive enqueues, and 8 consecutive dequeues. You will not need to modify the queue for this part, only the `main3.cpp`.

The makefile will compile this into an executable called `queue8`.

1.4 Optimizing programs and queues

In this final part, you will modify both the program and the queue. For the queue, you will implement the functionality to enqueue and dequeue 8 items at a time. The API for these two new functions are given in the `CQueue.h` file. They are `enq_8` and `deq_8`. The enqueue takes a point to a float array called `a`. It enqueues 8 floats starting at the inial location of the array, i.e. `e[0] - e[7]`. Dequeue is similar, it reads 8 values from the queue and stores them in `e[0] - e[7]`.

Be sure to check that the queue has enough elements to dequeue, and equeue before executing these functions. As you should find, the size check is a little more tricky than we discussed in class!

Now modify `main4.cpp` to take advantage of these new API calls. That is, the program should be similar to that of `main3.cpp`, however, instead of calling enqueue and dequeue 8 times consecutively, you can use the new API.

The makefile will compile this into an executable called `Queue8API`.

1.5 What to run

You should time each of your executables. I also suggested that you check each program for correctness.

1.6 What to submit

Submit the completed `SyncQueue.h`, `Queue.h`, `main3.cpp`, and `main4.cpp`. Please also include a report with a graph of your times and 1 paragraph explaining your implementation and 1 paragraph explaining your timing observations.

Your grade will be based on 4 criteria:

- Correctness: Do your schedules produce the right results.
- Conceptual: do your implementations use the concurrent queues in a way that balances work across the threads? Are your queues implemented correctly?
- Performance: do your performance results match roughly what they should.
- Explanation: do you explain your results accurately based on our lectures.

Challenge

- Generalize your `queue8` implementation to `queueN` where you enqueue and dequeue a variable amount of values. Identify how many values you can increase it to until performance start degrading
- Experiment with various queueu sizes. Do they make a difference? How big does the queue ever grow to?

2 DOALL Loop Parallel Schedules

We will cover the required material for this section in the May 11th lecture. We will consider different ways to parallelize DOALL loops, also known as a *parallel schedule*. Static work partitioning works well for DOALL loops where iterations take roughly the same amount of time. When loop iterations have more variation, it helps to use dynamic scheduling, e.g. workstealing. These dynamic strategies can either use a global worklist, or local worklists. Furthermore, the granularity of the tasks can be tuned.

Your assignment is to generate a several parallel schedules for the following loop:

```
void function(float *result, int * mult, int size) {
  for (int i = 0; i < size; i++) {
    float base = result[i];
    for (int j = 0; j < mult[i] - 1; j++) {
      result[i] = result[i] + base;
    }
  }
}
```

Notice that the outermost loop is safe to parallelize because the inner loop only reads and writes to arrays at index `i`. Each outer loop `i` computes the `mult[i]`-th multiplication of `result[i]` (assuming a greater-than-zero values in `mult`). You are not allowed to use the multiplication operator: it will be computed with repeat additions. Depending on the values in `mult`, there is potentially load imbalance across loop iterations. We will consider a linear load imbalance, where the amount of work for index `i` grows linearly with `i`. Larger values of `i` will take considerably more work than smaller values of `i`.

You will have 4 parts to this homework:

2.1 Static schedule

In `main1.cpp`, implement `parallel_mult` using a static partitioning. That is, each thread should compute the name number of `i` indices. You should split the work up in a chunking style, i.e. where thread `0` computes indexes 0 through `size/num_threads`; thread 1 should compute index `size/num_threads` through `2*(size/num_threads)`. This may be similar to your solution to problem 1c in homework 1.

You are responsible for creating the threads and joining them at the end. I have provided you data to operate on. Please use `results_parallel` as your results array, the `mult` array as the `mult` argument, the `SIZE` constant as the `size` argument. Instantiate the thread id as we've seen previously for SPMD programs. Use the `NUM_THREADS` constant as the number of threads.

The makefile will compute an executable called: `static` to run this part of the homework.

2.2 Global worklist workstealing schedule

In `main2.cpp` you will find a similar program to `main1.cpp`. However, instead of statically partitioning, you will use an atomic global counter to distribute work across threads. That is, you write the parallel function to use an atomic increment to get an index to calculate. For more information, see the global worklist material in the May 11th lecture.

Like `main1.cpp`, you are responsible for launching and joining the threads. The arguments should be passed in similar to `main1.cpp`.

The makefile will compute an executable called: `global` to run this part of the homework.

2.3 Local worklists workstealing schedule

In `main3.cpp` you will implement a local worklists workstealing schedule. First you should implement `IOQueue.h` as an Input Output Queue as discussed in the May 6 lecture. These queues need only support parallel enqueues, or parallel dequeues, but not both. You do not need to implement `dec_32` for this part of the problem. The `deq` should return -1 if the queue has no more elements.

The work items in this assignment are indexes to compute; i.e. they are integers between `0` and `SIZE - 1`.

In the main file, you should implement the `parallel_mult` function using a local workstealing schedule. The queues (one for each thread) are provided in the global variable `Q`. The function should provide the same results as the previous `parallel_mult` functions; I have deleted the initial code though because this implementation will be sufficiently different from the nested for loops.

You should first launch threads to initialize their local queues using the `parallel_enq` function. This function should be called in parallel in SPMD style. Each thread should enqueue an equal number of indices to compute. They should enqueue indices in a chunked style, similar to `main1.cpp`. Ensure the queues are initialized with enough space using the `init` function.

After the queues are initialized, join the threads in the main thread. Then call the `parallel_mult` function. Implement this function using a local worklists workstealing strategy. This strategy is discussed in the May 11 lecture. You will likely need the provided global atomic integer `finished_threads`.

The makefile will compute an executable called: `stealing` to run this part of the homework.

2.4 Task granularity

Your final implementation task is to increase the number of tasks (indexes) that are dequeued at a time. First, implement `deq_32` in `IOQueue.h`. This function dequeues 32 elements at a time and stores them in the argument array. It returns 0 if successful, or -1 if there are not 32 elements in the queue to dequeue.

The main function is the same as `main3.cpp`, with the exception that you should dequeue 32 elements at a time.

The makefile will compute an executable called: `stealing32` to run this part of the homework.

2.5 What to run

Please run each of your executable. Record how long each program takes to run.

Change the number of threads to 1 in `utils.h` and run `static` to get a single threaded measurement of runtime.

Change the number of threads to match how many cores you have on your machine.

2.6 What to submit

Turn in your completed: `main1.cpp`, `main2.cpp`, `main3.cpp`, `main4.cpp` and `IOQueue.h`.

Include a report that includes a graph of your runtimes. Explain your implementation in 1 paragraph. Explain your results in a another paragraph.

Your grade will be based on 4 criteria:

- Correctness: Do your schedules produce the right results.
- Conceptual: do your implementations use the concurrent queues in a way that balances work across the threads? Are your queues implemented correctly?
- Performance: do your performance results match roughly what they should.
- Explanation: do you explain your results accurately based on our lectures.

Challenge

- can you change the granularity of the global worklist? Does this make a difference?