CS380P: Concurrency

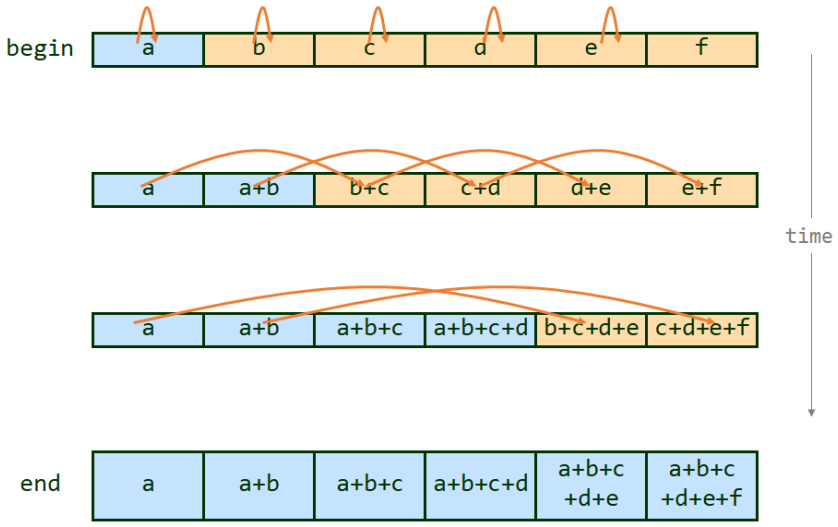
Lab #1: Prefix Scan and Barriers

The goals of this assignment are:

- To learn about parallel prefix scan algorithms
- Compare their performance using different synchronization primitives
- Gain experience implementing your own barrier primitive
- Understand the performance tradeoffs made by your barrier implementation

In this assignment, you will implement a work-efficient parallel prefix scan algorithm using the pthreads library and a synchronization barrier you will write yourself. We encourage you to think carefully about the various corner cases that exist when implementing this algorithm, and challenge you to think about the most performant way to synchronize a parallel prefix scan depending on its input size.

Background



Parallel prefix scan has come up a number of times so far in the course, and will come up again. It is a fundamental building block for a lot of parallel algorithms. There are good materials on it in Dr. Lin's textbook, and the treatment of it you can find on Wikipedia comes with reasonable pseudocode (see here). You may notice that prefix sum and prefix scan are in fact the same, with sum just being a single instance of the associative and commutative operator that scan requires. Note that *you will be implementing a work-efficient parallel prefix scan*.

Setup

You are given a quite big skeleton code (download here), along with a Makefile and two helper scripts (you could choose to completely ignore as long as you measure the elapsed time of your code the same way the skeleton code does). After understanding prefix scan you should get comfortable with the code given to you. It contains a working sequential prefix scan, try compiling and running it. There is a helper script that generates three inputs you are required to test against (1k.txt, 8k.txt and 16k.txt) and a simple input with 64 increasing numbers. As we mentioned before, prefix scan can be implemented with any associative and commutative operator. In this lab you will use an operator that is given to you (operators.cpp). This operator takes two numbers (like any other binary operator) and an extra parameter (-l) that modifies the amount of times the operator loops. In main.cpp you can switch between the operator you have to use (op) and simple integer adition (add), useful for debugging: scan_operator = add; or scan_operator = op; For each of the steps below, you will be comparing the performance of your prefix scan implementations for different parameters (argparse.cpp) over the input files that were generated. You are also given a simple python script that runs your code while changing paramaters (run_tests.py). You are again, free to change it as you like.

Sequential Implementation

The sequential version is called when the number of threads specified on the command line is 0 (`-n 0`). Note that using a single pthread (`-n 1`) is not the same, as it involves all the overheads of thread creation and teardown that would not be present in a truly sequential implementation. You can always test correctness of your parallel code by comparing to the sequential version.

Step 1: Parallel Implementation

Recall that prefix scan requires a barrier for synchronization In this step, you will write a work-efficient parallel prefix scan using pthread barriers. For each of the provided input sets, set the number of loops of the operator to 100000 (-l 100000) and graph the execution time of your parallel implementation over a sequential prefix scan implementation as a function of the number of worker threads used. Vary from 2 to 32 threads in increments of 2. Then, explain the trends in the graph. Why do these occur? From here: "A prefix sum algorithm is work-efficient if it does

asymptotically no more work (add operations, in this case) than the sequential version. In other words the two implementations should have the same work complexity, O(n)."

Step 2: Playing with Operators

Now that you have a working and, hopefully, efficient parallel implementation of prefix scan, try changing the amount of loops the operator does to 10 (-l 10) and plot the same graph as before. What happened and why? Vary the -l argument and find the inflexion point, where sequential and parallel elapsed times meet (does not have to be exact, just somewhat similar). Why can changing this number make the sequential version faster than the parallel? What is the most important characteristic of it that makes this happen? Argue this in a general way, as if different -l parameters were different operators, afterall, the -l parameter is just a way to quickly create an operator that does something different.

Step 3: Barrier Implementation

In this step you will build your own re-entrant barrier. Recall from lecture that we considered a number of implementation strategies and techniques. We recommend you base your barrier on pthread's spinlocks, but encourage you to use other techniques we discussed in this course. Regardless of your technique, answer the following questions: how is/isn't your implementation different from pthread barriers? In what scenario(s) would each implementation perform better than the other? What are the pathological cases for each? Use your barrier implementation to implement the same work-efficient parallel prefix scan. Repeat the measurements from part 2, graph them, and explain the trends in the graph. Why do these occur? What overheads cause your implementation to underperform an "ideal" speedup ratio?

How do the results from part 2 and part 3 compare? Are they in line with your expectations? Suggest some workload scenarios which would make each implementation perform worse than the other.

Inputs/Outputs

Your program should accept the following four command line parameters (already in the skeleton, if you choose to use it), with types indicated in parentheses, and the meaning of each parameter described in angle brackets:

1. -n <number of threads> (int) **(0 means sequential, not 1 pthread!)**
2. -i <absolute path to input file> (char *)
3. -o <absolute path to output file> (char *)
4. -l <number of loops that the operator will do> (int)
5. -s <use your barrer implementation, else default to pthreads barrier> (Optional)

The first line in the input file will contain a single integer *n* denoting the number of lines of input in the remainder of the input file. The following *n* lines will each contain a single integer/

The required inputs and a simple one can be generated by running the generate_input.py python script, but you are free to create your own tests for debugging purposes.

You shouldn't change the Makefile since our grading script will assume the binary is built at a specific place and name. Your solution should output the execution time in microseconds to stdout, and no other output. It is fine to produce other output on stdout as long as it is disabled by default, either at compile time (using macros) or at runtime (using an extra/optional command-line flag).

Deliverables

Submissions must be done on Canvas and you should include the following file in a tarball:

- A written report of the assignment in either plain text or PDF format. Please explain your approach, present performance results, discuss any insights gained, etc.
- All your source code required to build the binary, including the Makefile.

Notes

- Make sure that your implementations for parts 2 and 3 output a correct prefix sum for any number of threads.
- Please implement an inclusive prefix scan, as opposed to an exclusive one.
- You can modify/add any file you want. Just make sure the given makefile is unmodified and will compile your code correctly.
- Input size might not be a power of 2. You should handle that case.
- Do not print the length, or number of elements, when printing the output.

Please report how much time you spent on the lab.