

CS 380P ASSIGNMENT 1 REPORT

WORK

In this assignment we created a generalized parallel prefix framework similar, but not identical to the one found in the book. Unlike the book which used the parallel language Peril-I, we used pthreads which meant many modifications were needed.

The first major change is that fact that Peril-I is a compiler based parallelism language where specific loops are declared to be parallel and each index is executed via a separate thread. This is not as easily copied in pthreads, but we mirrored the basic idea by creating an equivalent number of threads. The loop data, which is simply implicit in Peril-I, must be passed manual as a struct in pthreads. This is used to set up thread specific data such as copying owned sections of the initial input data locally and setting thread ID.

The other major change is that we don't have full/empty memory constructs like Peril-I does. This isn't an issue for the first part of the program, but we needed to work around this in the second portion to induce blocking when reading an "empty" segment or writing to a "full" segment. Without this manual checks, we'd have parallel data hazards (WAR) because valid data could be overwritten by fast threads before it was read.

As part of our assignment, we were required to implement two different parallel prefix operations

PROBLEMS

Our main source of difficulty was following the algorithm provided in the book too closely. We expected it to be hypothetical as it is written in a made up language, but we didn't expect the deviation that was required. The first portion (pulling up the values) worked basically without issues, but pushing the values back down didn't seem to work despite all our best efforts. Even with locks, we would have randomly interleaved data leading to us to conclude that their obviously must be race conditions on something we weren't catching.

To isolate the problem, we separated the two portions of the problem running each one multiple times and diff-ing the results. While theoretically, we could possibly be missing a few rare race conditions in this method, it is unlikely because we were running this on a true parallel machine where threads DO run concurrently and are likely to be mixed randomly. After this testing, we concluded that the first portion of the code was not the issue as it was stable. Given this assumption, we could treat its output (and input into our second half) as deterministic with the exception of where the barrier between the first portion and the second portion of the program.

To fix this, we assumed that either the algorithm must be fault or we were not synchronizing correctly. We set about hand checking our executions and discovered our algorithm was unstable. This was corrected by altering the algorithm to remove this instability.

CORRECTNESS/TESTING

Like most parallel algorithms, ours is sensitive to race conditions unless these are explicitly handled and prevented. We achieved correctness by adding locks, using local copies of data, and buffering writes to prevent RAW hazards. We feel confident that these techniques prevent most errors.

To test our code, we did the only thing we felt reasonable: execute a variety of data (at multiple processor counts) and compared the output (using diff and printf) to multiple iterations to check for both accuracy and consistency.

EXAMPLES (SEE DATA.TXT)

FIRST INDEX	LAST INDEX
FINDS THE FIRST INDEX OF ALL NUMBERS	FINDS THE LAST INDEX OF ALL NUMBERS
ohmu:~/classes/cs380p/assign1> fid data.txt First index for each number [1] found first @ index 3 [2] found first @ index 0 [3] found first @ index 2 [4] found first @ index 9	ohmu:~/classes/cs380p/assign1> lid data.txt Last index for each number [1] found last @ index 19 [2] found last @ index 12 [3] found last @ index 17 [4] found last @ index 16

The above data was generated by a simple Perl script with inputs for K and N. We have found from experience that it is often the small parallel cases which fail obviously, so this case is displayed here.

Data is a simple ASCII file with one integer per row.