C Library for fork-join style parallelism

Ayush Agrawal (ayushagr)

[I couldn't find anyone else to work with. I did make a post on Piazza to find a partner but got no response to that].

https://www.andrew.cmu.edu/user/ayushagr/

I plan to write a C library that provides support for fork-join pattern of parallelism to:

- compare continuation stealing with child stealing (average length of work queues, synchronization overhead, complexity of implementation, etc)
- understand the intricacies of managing distributed work queues and work stealing

Quoting wikipedia, "... the fork-join model is a way of setting up and executing parallel programs, such that execution branches off in parallel at designated points in the program, to join (merge) at a subsequent point and resume sequential execution. Parallel sections may fork recursively until a certain task granularity is reached. Fork-join can be considered a parallel design pattern. By nesting fork-join computations recursively, one obtains a parallel version of the divide and conquer paradigm."

In class we studied cilk. Originally developed at MIT, it has now been adapted as an open standard (in GCC, Intel ICC). Cilk Plus extends the C++ runtime and it always run spawned child (continuation stealing) first. What I would like to do is first come up with a similar framework in C but which supports both types of stealing and then see for myself why would they prefer one over the other (a quantitative comparison).

Implementing distributed scheduling, work stealing and maintaining a pool of threads would definitely be a challenge. To avoid bottlenecking on a single work queue, I will have work queue for each thread. Design choices will include choice of synchronization primitives, continuation vs child first scheduling, choosing a victim to steal, etc.
RESOURCES

I will build the C library on top of POSIX threads so that it can be used on any POSIX compliant system. I will be going through the publications that came out of cilk for reference. The ones that I have currently planned to read are:


GOALS AND DELIVERABLES

Plan to Achieve: Have a working library that can be used to model fork-join pattern for any underlying problem. Having implemented both types of stealing, come up with a comparison of the two across 3 different algorithms. Be able to quantitatively talk about the overhead of maintaining distributed work queues and what all optimizations went into implementing the system. I anticipate that the average work queue length will be smaller in case of continuation stealing and moving from a single work queue to a distributed work queue should improve the performance significantly.

Hope to Achieve: In case I have some time left, I could either choose to extend the library to support remote threads as well or change the library to run over the Pebbles kernel (part of the Operating Systems class) instead of using POSIX threads.

Worst Case: Have a working library but not enough data (generated from runs of different algorithms implemented using my library) to compare the two stealing techniques. If I get stuck, I might end up choosing a simple technique for stealing work (e.g. from the tail of a work queue) instead of doing something fancy like random victim, etc.

Demo: For the demo, I will be demonstrating how you could easily take a recursive problem that uses divide and conquer and express it using the library implemented and show the speedup obtained (as a function of no of threads). Other than the speedup, I will compare the two stealing approaches based on total execution time, time spent in computation, time spent in synchronization, time spent waiting, etc. Other metrics for comparison will include the average size of the work queues (i.e. space complexity), etc.

PLATFORM CHOICE

I am choosing to implement the library in C because first of all it will give me more control over my implementation and secondly if it works for C, it will work for C++ naturally. Given that most parallel applications are written in these languages, this will make my library more useful and I
will be able to use it in my future projects. I will be testing and profiling my framework on the GHC machines. The GHC machines have 8 physical cores (16 logical ones), so I will be able to observe good speedup with the increase in number of threads - if I was working on say a machine with just 2 cores, the speedup “might” flatoff real early.

**SCHEDULE**

- Nov 1 - Nov 7: Read up the literature specified above
- Nov 8 - Nov 14: Implement single work queue continuation first
- Nov 15 - Nov 21: Extend to distributed work queue with work stealing
- Nov 21 - Nov 27: Extend to implement child first
- Nov 28 - Dec 4: Rewrite 3 divide and conquer algorithm to make use of the library
- Dec 5 - Dec 8: Get the comparison stats and prepare the final report
- Dec 9 - Dec 10: Prepare for and present the work done so far