GPUMC Assignment 2: Kentucky's Line Extrusion Orderer

Implementor's Notes

Hank Dietz
Department of Electrical and Computer Engineering
University of Kentucky, Lexington, KY USA
hankd@engr.uky.edu

ABSTRACT

This project involved modifying a serial C program that uses a genetic algorithm to reorder extrusion of line segments to minimize print time (total travel time between line segments). The idea is to speed-up the program by using OpenMP.

1. GENERAL APPROACH

The key to getting speedup is to parallelize things at the highest level possible. Here, there were two convenient spots:

- The population initialization loop. It turns out that `greedy()` is one of the most expensive operations, so parallelizing execution across multiple copies of it yields good speedup.
- The loop over all "generations" in the genetic algorithm. This offers a huge amount of parallelism if the population is large, but fundamentally hits two problems. First, the serial steady-state GA only replaces one population member at a time, but this will replace multiple ones simultaneously, so we'll need to ensure the population members being worked on are disjoint. Second, there is a subtle change in the search statistics because disjoint sets of members being operated upon means that a newly-created population member can be directly involved in no more than one of the `nproc` simultaneous creations of new members... this will slow convergence.

Both of those parts of the code simply turn into OpenMP `parallel for` constructs, however, there are a few things that need to be dealt with for shared access:

- A lot depends on being able to `mkorder()` in parallel, so we need to ensure threads use a private (local) order buffer.
- The standard `rand()` isn't thread safe, so we need to use something else... here, `rand_r()` with `rseed[iproc]` for its state and inital values generated sequentially calling `rand()`. It turns out that `rand_r()` is not very random in the low bits, so for `RANDPLACE`, I divide the value by 13 to help randomize the low bits. Using a prime population size would also help randomize.
- The update of the best value found so far is protected by an OpenMP lock: `bester`.
- To avoid deadlock, each thread needs to pick (and claim) all the population members for making a new member in one shot. This is done by `claim3()` and they are released by `unclaim3()`. The array `claimed[]` tracks which population members have been claimed, and the OpenMP lock `claimer` ensures one one claim is processed at a time.

Note that both updating the best and claiming members to work on are done in ways that try to minimize the work done while locked.

2. PERFORMANCE

Running on 4 procesors, typical speedup is between 2X and 3X.

The final schedule is now sent to `stderr`.

3. ISSUES

The modified code will not work without `-fopenmp`.

Parallel results obtained were often slightly inferior to the sequential version because of the aversion to reuse of the best. Perhaps `claim3()` should prefer picking `abest`? To better understand how the search worked, I replaced the `VERBOSE` tracking with logic that tracks the history of each population member in the array `by[][]`. This revealed that new best are rarely derived from Random starts, and almost never combined Original, Greedy, and Random.

There was a bug in `rotate()` in the distributed version of `kleo.c` that made `a` a rotation of `a`, not of `b`. It was fixed and a comment inserted.

ACM ISBN N.A..
DOI: N.A.