Exercise 3 20P

submission date: 05 Dec (23:59)

This exercise is made to show the potential of OpenMP to parallelize simple codes. OpenMP is always an option if all threads are essentially performing the same code. This experiment is less structured than previous exercises. It is more about experimenting with the available methods than following any specific interface. Some of the mentioned effects have little or no influence on consumer hardware (single-socket-cpus), i.e., find out for yourself what has influence on your machine.

Tip: If your private computer has less than 4 threads (preferably 8), use a computer from the ATIS, or ask us for an account on our local compute servers.

3.1 Microbenching Different Access Patterns (6)

The goal is to sum up an array of random numbers. Create an array with n random numbers ($\approx 1\,000\,000\,000$). Use OpenMP to parallelize a `for`-loop that sums up all numbers (subtasks a-d can all be solved in one executable).

Note: You can use `std::mt19937` to generate random numbers. Use a preset seed to make your tests reproducible.

a) Atomic contention (1)

Use a single atomic variable to add up all numbers.

b) False Cache Sharing (1)

Use an array of local counters. Each thread adds to its own counter, and afterwards the counters are combined to the global count. Do not create any separation between the counters (they should share a cache line).

c) Fixed (1)

Use a similar construction as before (in b), but make sure that no two local counters are stored in one cache line.

d) Automatic/OpenMP (1)

Use OpenMPs `private` annotation to create the local variables (use one variable and declare that as private in the OpenMP pragma. Each thread will have and add to its own variable).

e) Evaluation (2)

Test each of your implementations with a varying number of threads and display your results (do not use more threads than your computer has available, but use hyperthreading if available).

3.2 Parallel Connecting Forest Computation (14)

In this task we want to compute a spanning forest of an undirected graph, which is only given as a vector of edges. To compute the forest, we use a streaming algorithm that never needs to store all edges at once (only tree edges are stored, each other edge is considered exactly once).

The algorithm works similar to Kruskal's algorithm, but without first sorting the edges. (1) For each edge we have to check if, together with the already chosen edges, it constructs a cycle. (2) Safe edges are selected and added to a preliminary graph data-structure. (3) Afterwards, we want to compute the parent array of the resulting graph. Every step of this algorithm can be done in parallel.

(1) To check whether an edge introduces a cycle, we use a union-find data structure (one n-sized array storing NodeIDs and one n-sized array for the rank information). When the union operation is implemented thread-safely, then several threads can filter/choose edges concurrently. Thus, parallelizing this part of the algorithm can be done straightforwardly, by distributing the edges between threads. Use a single array to store all edges that you have filtered (size n). The counter for the number of currently stored edges might be a point for contention (potential for optimization).

(2 and 3) The forest has to be returned as a parent array. But step (1) only results in a list of chosen edges. To build a tree from these edges, you can first compute an adjacency array or adjacency list (see exercise 2) in parallel (do not use more than $O(n)$ memory). Then you will have to use breadth first searches from each of the union-find represen-

tatives to compute the parent array. Do not try to parallelize one single BFS, instead execute multiple BFS in parallel.

The functionality (`dynamic_connectivity.hpp`) should be implemented in a way that allows future calls to the `addEdges` function. Future calls can add to the existing forest by unifying some of the existing trees. Note that this changes the parent array, because edges have to be redirected to the common root. Therefore, you should store the array of chosen edges, and keep the union-find datastructure.

The structure of this exercise is a little different than the previous implementations. You will implement your algorithm in a `.cpp` file. This allows you to produce different implementations of the same header file (please make sure that the final .cpp file is named `dynamic_connectivity.cpp`). We have a small correctness test supplied for you, however, this test might not be sufficiently large to find specific edge cases of your algorithms. You will have to program very carefully potentially implementing your own additional tests. We will not check your correctness tests for plagiarism, instead we would encourage you to share tests among yourselves and also check them into your repositories (However do not include any large files into your repositories, otherwise you will quickly run into problems with the SCC memory quotas). Very common bugs could include union find data structures that create circles, or splitting of already joined subtrees (both might be caused by race conditions). Another common problem could be outdated rank or size information (somewhat less problematic because it does not impact the correctness).

a) Sequential implementation (3)

Build a sequential implementation. This implementation should not have any overheads due to locking/atomics used by the parallel variant. It is possible to merge some steps of the algorithm described before.

b) Thread-Safe Union-Find Filtering (1)

Implement a thread-safe variant of the union-find datastructure. Lock both roots whenever you unify two trees (You may use `std::mutex` for the locking since OpenMP does not offer anything equal).

c) Path Compression (1)

Implement path compression within your union-find data structure.

d) Union by Rank (1)

Implement union-by-rank within your union-find data structure (alternatively use union-by-size).

e) Parent Array (2)

Compute the parent array in a parallel manner (steps 2 and 3 mentioned before). *Tip: Maybe you can do some precomputation during the previous part of the algorithm.*

f) Lockless Variant (2)

Now we want a lockless implementation; is the same algorithm still possible? Are there any changes which have to be made? Are both optimisations still possible (path compression and union-by-rank)? Implement a lockless variant that is as good as possible.

Tip: The problem with faulty lockless implementations is that potential errors can happen very rarely. Therefore, correct test results might not necessarily indicate a correct algorithm, especially when it is run with fewer threads. Sometimes running an algorithm with more threads than your machine has will lead to weird scheduling effects, which can help you find broken edge cases.

g) Time Evaluation (4P)

Test your algorithms with a varying number of threads. Present several reasonable performance measures like time and speedup. Implement this test in an additional cpp file (Do not use correctness.cpp).

In addition to the natural metrics (running time, speedup, and efficiency) find at least one interesting problem-specific metric and present some measurements. You could count the number of times certain things happen…

Test your algorithms on multiple graphs. You can construct graphs using the graph constructor that is placed in the repository (see README for details). Choose different sizes of graphs, and different structures.