Alan Mazankiewicz Freestyle Assignment

Topic: Approximated Maximum Weighted Matching in Weighted Complete Undirected Graphs

In my assignment I am going to implement sequential and parallel approx. solutions to the general weighted matching problem in complete undirected graphs. A matching in a graph is a subset of edges M such that no two elements of M have common endpoints. In an edge weighted graph, the objective is to compute a matching such that the sum of the weights of the elements in M is maximum. As the runtime for finding optimal solutions is to high for big graphs numerous approx. solutions have been proposed. In particular I am going to implement the sequential and parallel version of the Locally Dominant Algorithm as described in [1] as well as the sequential and parallel Suitor Algorithm as described in [2] (without the Heavy Matching quality optimization described in Section 4). For the evaluation I will make use of the GPA [3] Algorithm which according to [1] and [2] usually results in the highest quality approx. matching. I will use the implementation provided by [3]. It should mainly work as a benchmark for evaluating the quality of the matching. The data for the evaluation will either be generated using an external generator or some existing dataset. As [1] requires a concurrent queue I will either use the implementation from exercise 4 or some other external implementation. **Edit: After talking with Tobias I have changed the following: I will not compare myself with the GPA implementation but I will implement the sequential quality improvement ("Heavy Matching") of suitor [2] Section 4 and compare the quality against it instead.**

Note: In the following Pointer does not necessarily refer to a C++ pointer but to any form of pointing to an element be it an iterator or size_t that works as an index of an array etc.

Exercise 1: Locally Dominant Algorithm [1] (6P)

1.1 Graph Data Structure (1P)

For the Algorithm described in [1] (and also later in [2]) a weighted undirected graph data structure will be needed that can load a graph form a file. In this data structure a vertex has to be able to point to its matching neighbor. Therefore, an array of pointers has to be implemented. You can either store that array as a member of your graph or outside in the matching algorithm. Make sure that the edges are randomized in their weight order either by using an appropriate dataset or by randomizing them during construction. For [2] slight modification will be needed described in the respective part.

1.2 Sequential Locally Dominant Algorithm (3P)

Implement the sequential version of the Locally Dominant Algorithm as described in [1] Section 3.1 (Algorithm 1 and 2).

For this you will need a queue Q that consists of unprocessed matched vertices and a vector/array candidate that contains a pointer of the current-heaviest neighbor of each vertex (see 1.1). In Phase 1 for a given vertex s, all of its neighbors are scanned to find the current-heaviest neighbor that has not already been matched. The identity of the heaviest neighbor for each vertex is stored in vector candidate. After setting the candidate mate or vertex s, say to vertex t, we check whether the candidate mate for t is also set to s: candidate[candidate[s]] = s. If true, we have found a locally dominant edge. We add this edge to M, and the two vertices s and t to the queue. In Phase 2, we iterate until the queue becomes empty. During each iteration of the while loop, we process vertices matched in previous iterations while adding new vertices to the queue that become eligible as edges get matched. The algorithm will terminate when the queue becomes empty.

1.3 Parallel Locally Dominant Algorithm (2P)

Implement the parallel version of the Locally Dominant Algorithm as described in [1] Section 4.1 (Algorithm 3) using std::thread().

Similar to the serial version, we divide the parallel algorithm into two phases. Phase 1 of the algorithm consists of two parallel sections. Each section iterates once over all of the vertices in a graph. While the first section finds a candidate mate for each vertex, the second section checks for locally dominant edges and adds them to the matching. Phase 2 of the algorithm is executed several times by iterating over all the vertices in the queue until no new edges get matched (the queue becomes empty).

Exercise 2: Suitor Algorithm [2] (7P)

2.1 Sequential Suitor (3P)

Implement the sequential non-recursive suitor algorithm as described in [2], Section 3 (Algorithm 2). You can basically use the graph from 1.1, however remember to adjust the graph for a second pointer array (ws()).

Unlike the algorithm by [1] we do not make use of a central queue for storing the dominant edges found so far- We store for each vertex which (if any) vertex is pointing to it using a variable suitor(). To easily be able to determine the value of w(u, suitor(u)) (local heaviest weight) we store this value in a separate value ws(u) initially set to 0. We use a while loop to iterate over all vertexes and find for each vertex a partner vertex (with heaviest weight) and set suitor() for each edge accordingly.

2.2. Parallel Suitor using Locks (2P)

Adjust the sequential algorithm from 2.1 to work in parallel using locks as described in [2], Section 3 (p. 4f.) and using std::thread().

Parallelism is achieved by running over all vertices in parallel. Suitor() and ws() are shared variables for which threadsaftey has to be ensured. This is done by using |V| locks when accessing suitor() or ws() of a given partner vertex and checking the invariant after each acquire, otherwise we search for a new partner.

2.3 Parallel lock-free Suitor (2P)

Now implement the parallel lock-free variant of the suitor as described in [2], Section 3 (p. 5.). Adjust your graph data structure accordingly, that is get rid of the heaviest weights array and the lock array.

Instead the suitor() array should hold direct pointer to the heaviest weights in the weight array. Updating suitor() has to take place using Compare and Swap thus make it atomic.

Exercise 3: Grand Evaluation (7P)

~~**3.1 GPA Implementation (1P)**~~
~~Adjust the GPA implementation [3] to you interface~~
3.1 Sequential Suitor Heavy Matching Implementation (1P)
Implement the Sequential Suitor Heavy Matching Implementation as described in [2] Section 4.

3.2 Sequential (2P)
Evaluate your properly working sequential implementations in term of runtime against each other and against ~~GPA~~ Suitor Heavy Matching.

3.3. Parallel (2P)
Evaluate your properly working parallel implementations against each other in terms of runtime, absolute speedup and efficiency.

3.4 Quality (1P)
Evaluate the matching quality of all implementations against each other. Does parallelism reduce the quality of the matchings?

3.5 Tradeoff Evaluation (1P)
Evaluate the tradeoff between the speed of the parallel implementations against the matching quality of ~~GPA~~ Suitor Heavy Matching by plotting for each algorithm quality against speed. Is there a dominant implementation?

Reference
[1] M. Halappanavar, J. Feo, O. Villa, A. Tumeo, and A. Pothen, "Approximate weighted matching on emerging manycore and multithreaded architectures," Int. J. High Perf. Comput. App., vol. 26, no. 4, pp. 413– 430, 2012. (https://journals.sagepub.com/doi/full/10.1177/1094342012452893)

[2] Manne, Halappanavar: Suitor algorithm for 1/2-approx. edge weighted matching, IPDPS 2014. (http://www.ii.uib.no/~fredrikm/ipdps2014.pdf)