Project information

- Project name: parallel maximum flow
- Team members: Siyuan Chen, Xinyue Yang
- URL: `https://xinyue-yang.github.io/parallel-maxflow/`

Summary

We are going to parallelize maximum flow algorithms (specifically Dinic's and possibly push-relabel) under the shared address space model and analyze their performance on GHC and PSC machines.

Background

Flow in general is widely used to model constrained resource distribution problems. A variety of flow problems exist, e.g., maximum flow, minimum cost maximum flow, etc., and for each several sequential algorithms exist.

We choose to focus on maximum flow and the corresponding Dinic's algorithm. If time permits, we will also attempt to parallelize another algorithm for maximum flow—push-relabel.

The input to the maximum flow problem consists of a directed graph G; among the vertices are the source s (which has in-degree 0) and the sink t (which has out-degree 0); each edge also has an associated nonnegative capacity $c(u, v)$. The output to the problem is an assignment of flow to each edge such that the following two constraints are satisfied.

- Capacity constraint: the flow for each edge is nonnegative and at most the capacity of the edge.
- Flow conservation: for every vertex except for the sink and source, the incoming flow equals the outgoing flow.

Specifically, we will be focusing on integer flows (all capacities are integers) for simple graphs (there is at most one edge going from one vertex to the other).

Dinic's algorithm solves the maximum flow algorithm by repeating the following two steps.

1. Run BFS from the source to assign a to-source distance to each vertex; if the sink is not reachable, we have found a maximum flow.
2. Run DFS from the source repeatedly on the layer graph, pushing as much flow through as possible, augmenting the residual graph and marking edges as saturated when appropriate; when the sink is no longer reachable, we have found a blocking flow; throughout this process, the maximum flow is continuously updated.

In later reports, we will provide a more in-depth explanation of Dinic's algorithm and clarify the technical terms above, e.g., saturated edges, residual graph, layer graph, blocking flow, etc.

Challenges

In general, it is difficult to design performant parallel graph algorithms due to nonregular memory access patterns and complex dependencies. In this case, we find Dinic's algorithm to be an interesting candidate to parallelize since it consists of both computation that seems more straightforward to parallelize (BFS in the first step) as well as computation that seems nonintuitive to parallelize (DFS in the second step).

For the BFS, we plan to process each vertex in the frontier in parallel and construct the next frontier using some sort of efficient set union mechanism.

For the DFS, we will experiment with having multiple processors each traverse the layer graph and find augmenting paths, and will try to find smart ways to resolve conflicts (intersecting paths). The nice thing is that for Dinic's, the blocking flow for the second step doesn't have to be exactly blocking, and so we can also potentially add a tuning parameter to adjust how much effort the algorithm puts in here before going to the next iteration.

Besides overcoming challenges throughout the designing and implementing of the algorithms, the project will also be a significant learning experience for us in terms of profiling and analyzing the performance of parallel algorithms. We plan to comprehensively examine the algorithms' cache efficiency, memory sensitivity, speedup and scaling, etc., and plan to produce informative and interpretable graphs for each performance characteristic.

Resources

To profile and analyze the performance of the algorithms, we will be using the 8-core GHC machines and the up-to-128-core PSC machines. At this moment, we do not recognize a significant benefit in testing and profiling on other architectures, but we are of course open to advice and feedback in this area.

We will be implementing the algorithms as well as the testing and profiling framework from scratch in C++. We anticipate frequently referencing 15-451 Algorithm Design and Analysis's lecture notes on Dinic's algorithm.

Goals

We plan to achieve the following goals.

- Implement sequential Dinic's algorithm.
- Parallelize the first step (BFS) of Dinic's algorithm using parallel set union.
- Parallelize the second step (DFS) of Dinic's algorithm to a reasonable degree using fine-grained locking.
- Profile and analyze sequential and parallel Dinic's algorithm:

  - on different architectures for different core counts: GHC, PSC;
  - for different graph sizes; and
  - for several different graph categories: dense, sparse, high-degree, low-degree, etc.
- Provide a deep understanding of the bottlenecks and weaknesses of the current parallel implementation of Dinic's algorithms.

We hope to achieve equivalent goals for the push-relabel algorithm.

Deliverables

We have the following deliverables.

- source code for the sequential and parallel versions of the algorithms;
- source code for the testing and profiling frameworks;
- test cases for different graph sizes and categories;
- performance data for the different experiments;
- graphs explaining the performance data; and
- (potentially for the demo) animated toy graphs that show how the two steps are parallelized across processors, especially for the multi-head DFS.

Platform

As previously mentioned, we will be implementing our algorithms using C++ and profiling them on the GHC and PSC machines. We believe these are quite reasonable, self-explanatory choices since C++ has great support for the OpenMP framework which we aim to utilize (since we are operating under the shared address space model). Regarding the machines, we will just stick to the default for now but are definitely open to advice and feedback.

Schedule

- Week 1 (Mar. 25th–Mar. 31st): literature review, brainstorming, outlining
- Week 2 (Apr. 1st–Apr. 7th): sequential implementation, testing framework, test cases
- Week 3 (Apr. 8th–Apr. 14th): initial parallel implementation, initial profiling, improving testing framework, more test cases
- Week 4 (Apr. 15th–Apr. 21st): milestone report, improving parallel implementation, more profiling, initial data analysis and visualization
- Week 5 (Apr. 22nd–Apr. 28th): more profiling, more data analysis and visualization
- Week 6 (Apr. 29th–May. 5th): final data analysis and visualization, final report, poster