— *Project* —

Parallel Algorithms and Programs

Hugo SALOU

Throughout this project, we tested the implementations on about 500 small graphs, and comparing with the expected flow value for each one. This expected value was given by the `networkx` Python library. These small testing graphs had at most 14 vertices, and between 2 and 5 processors (or 1 when testing the sequential algorithms). They were able to detect errors quickly. The script `test_flow.py` runs all these tests in a few minutes (or, equivalently, `make test`).

After we were confident the implementations of Edmonds-Karp's and Dinic's algorithms worked without issues on small graphs, we started scaling the graphs to tests the performance. In the next section, the testing strategy will be presented, how we sample the graphs for performance-testing.

1 Sampling Random Graphs.

To test, two samplers were used: one for sparse graphs, and one for dense graphs. All of this is managed in the `bench_flow.py` script. Table 1(b) gives the random distributions used for graph sampling, and table 1(a) gives us the MPI settings used. We write $\mathscr{U}(X)$ for the uniform distribution on set X, and $\mathscr{E}(\lambda)$ for the exponential distribution with parameter λ. The term δ in the sampling for the number of vertices N is used to ensure that P divides N.

Table 1 | *Parameters Sampling for Performance Testing*

(a)

MPI Parameters	
Number of Processors P	$P \hookleftarrow \mathscr{U}(⟦10, 150⟧)$
Link Bandwidth	100 Gb/s
Link Latency	10 µs
Simulation node compute power	100
Real machine compute power	100
Topology	complete

(b)

Type of Graph	Sparse Graph	Dense Graph
Number of Nodes N	$N \hookleftarrow \lfloor \mathscr{E}(1/10\,000) \rfloor + 10 + \delta$	$N \hookleftarrow \lfloor \mathscr{E}(1/1\,000) \rfloor + 1\,000 + \delta$
Number of Edges M	$M \hookleftarrow \mathscr{U}(⟦n-1, \min(\lfloor n(n-1)/2 \rfloor, 50N)⟧)$	$M \hookleftarrow \mathscr{U}(⟦\lfloor n(n-1)/3 \rfloor, \lfloor n(n-1)/2 \rfloor⟧)$
Maximum capacity C	$C \hookleftarrow \mathscr{U}(⟦1, 100⟧)$	

After sampling these values, we choose uniformly the indices of the source s and target t in $⟦1, N⟧$. Using `networkx`, we sample uniformly a graph G with N vertices and M edges which is weakly-connected, and contains an st-path. For each edge in this graph, we assign a random capacity following the uniform distribution on $⟦1, C⟧$.

Figure 1 (page 2) shows the number of edges and vertices for each sampler. Table 2 (page 4) also gives statistical results.

Also, when testing the performance of the implementations, we also check that we have the correct flow value (assuming `networkx`'s is correct), in an effort to ensure correctness of the implementation.

In the end, we sampled around 500 sparse graphs, and 150 dense graphs.

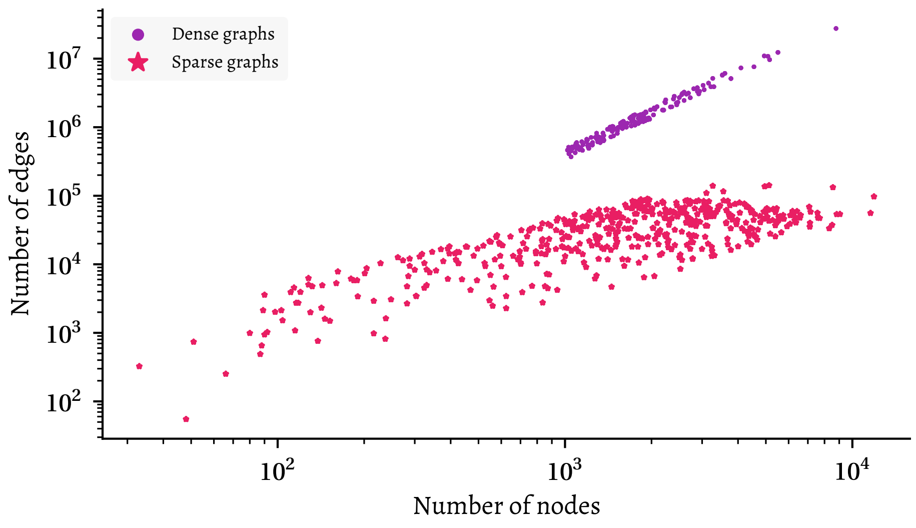


Figure 1 | *Random Graph Sampling (sparse* vs. *dense)*

2 Performance Testing of Implementations.

We decide to use the following metrics to track the performance of our implementations:

- ▷ the (simulated) execution time T_P with P processes;
- ▷ the parallel speedup with P processes $S_P := T_P / T_1$;
- ▷ the "theoretical" speedup $T_P / C_P(N, M)$ where $C_P(N, M)$ is the time-complexity of the algorithm.

The first metric allows us to compare the two algorithms' performance (as comparing the speedup would only tells us how our implementation "scales"). The second metric allows us to look at how much parallelization speeds up the flow computation. The last metric allows us to look at the "correctness" (in terms of time-complexity) of our implementations, as it should stay bounded. For Dinic's algorithm, we set $C_P(N, M) = N^2 M$ and for Edmonds-Karp's we use $C_P(N, M) = NM^2$.

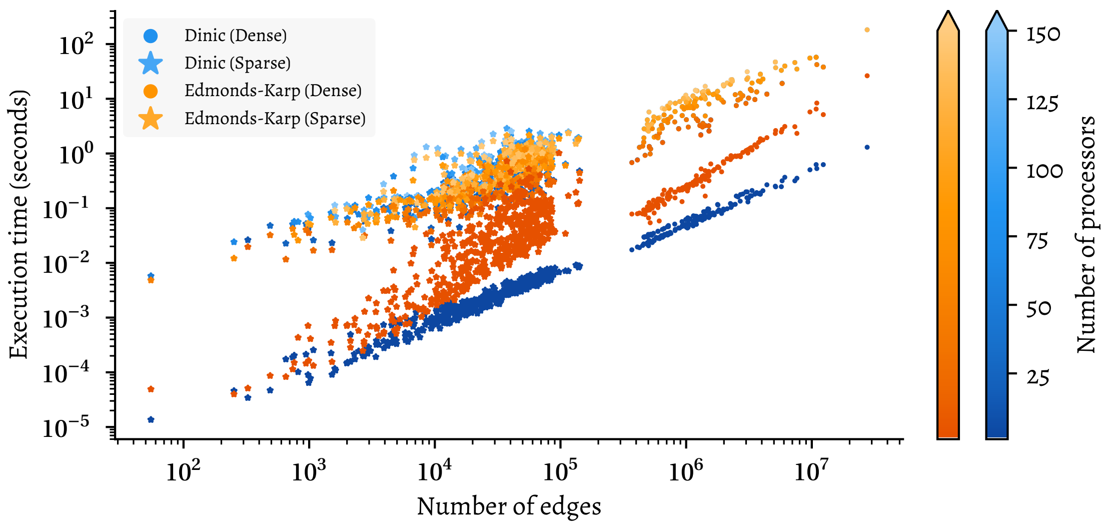


Figure 2 | *Execution time for each implementation*

Figure 2 show the execution time for each algorithm on the two kind of graphs sampled (the left part only contains sparse graphs, and the right one only dense graphs). Parallelization does not help a lot when dealing with dense graphs, but it could rivals the performance of sequential algorithms on some sparse graphs. The main bottleneck with parallelization for dense graphs is that the algorithms already have a high-enough time-complexity, so "spreading" data between multiple processes adds even more overhead.

In retrospect, allowing the use of up to 150 processors did not help with the time complexity: with the log-log scale, adding a few more processes does not improve by a lot the runtime. This makes sense as, when we increase the number of processes, we also dispatch the graph's data in more places so, at a certain point, adding more processes will only add data-retrieving-time, and not parallel-computation-time.

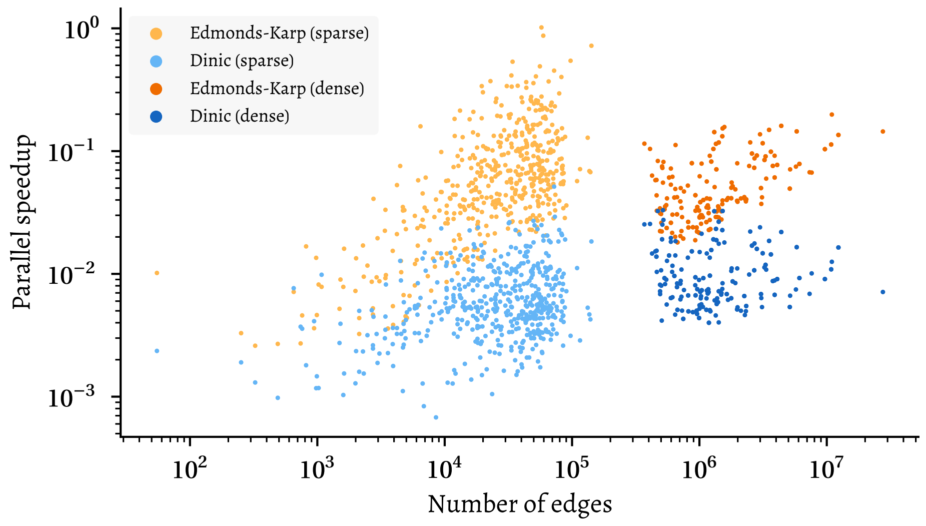
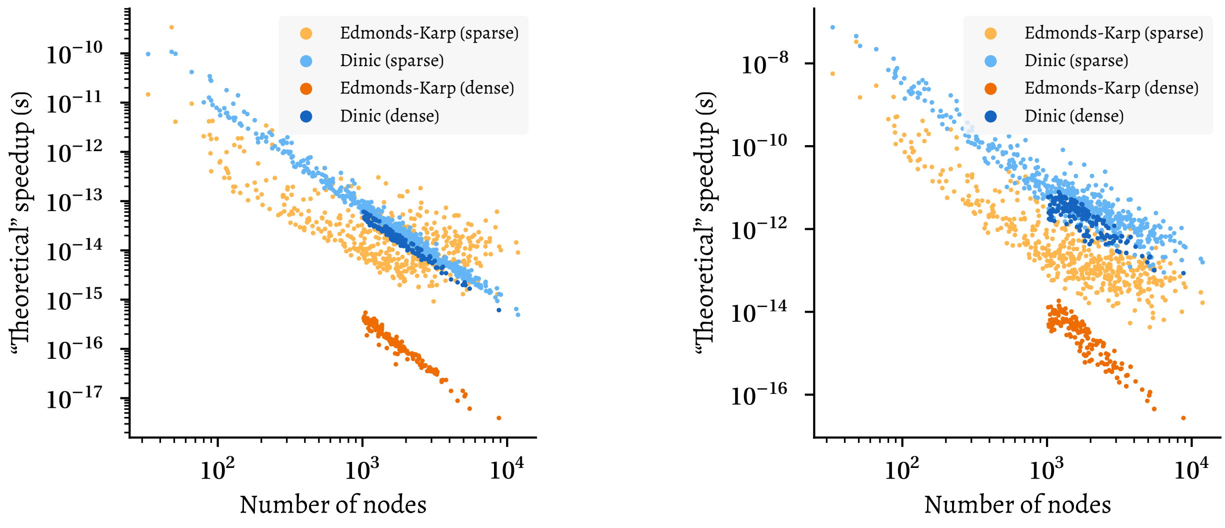


Figure 3 | *Parallel speedup*

In terms of parallel speedup (*see* figure 3), we did not gain a lot of performance: this is possibly due to a suboptimal parallel implementation of the two algorithms, or an overhead too large due to the link latency and bandwidth. An interesting experiment would be to vary those parameters instead of relying on those in table 1(a), maybe figure 3 would look a lot different. Due to a lack of time, we were not able to do this experiment: simulating these implementations took about 40 hours.



(a) Sequential Implementations (b) Parallel Implementations

Figure 4 | *"Theoretical" speedup*

Looking at the "theoretical" speedup (*see* figure 4), we see that our implementations have the expected time-complexity, or at least for the graphs sampled. We can also see that, for dense graphs, the bound on Edmonds-Karp's time-complexity is "looser" than the one on Dinic's time-complexity.

3 Conclusion.

Our implementations of Dinic's and Edmonds-Karp's algorithms is correct, has the required time-complexity, yet the parallel implementations is slightly worse than the sequential one, which is unexpected, but optimizing parallel implementations of algorithms is not as straightforward as optimizing a sequential algorithm.

Reproducing tho results should be as effortless as running `make bench`, choosing the implementation whose performance will be tested, and the graph sampler used (sparse or dense). It'll then create the tests, the hostfiles and platform files for MPI testing, and start running the tests. A CSV file will be produced containing the simulation data. Assuming all benchmarks have been done, the Python notebook `plots.ipynb` will produce the plots used in this report. This Python notebook is especially useful for further analysis of the data. Expect a few hours of waiting to get results (though you can stop the script when simulating at any moment, and the already-computed results will be put in the CSV files).

End of report.

Type of Graph		Dense Graph	Sparse Graph
	Count	737	2 000
Vertices	*mean*	1 996	2 205
	st. dev.	1 023	1 881
	min	1 020	33
	median	1 705	1 722
	max	8 772	11 890
Edges	*mean*	2 059 699	35 460
	st. dev.	2 813 298	25 470
	min	369 773	55
	median	1 187 210	32 434
	max	27 584 196	141 898
Processors	*mean*	47	40
	st. dev.	49	49
	min	1	1
	median	31	6
	max	149	150
Flow	*mean*	20 677	242 733
	st. dev.	16 234	257 423
	min	458	55
	median	17 660	148 884
	max	98 396	1 504 156
Time (ms)	*mean*	7 344	267
	st. dev.	13 589	410
	min	17	0
	median	3 202	78
	max	182 576	2 863

Table 2 | *Statistical results*