Parallel Route Planning with Time-Dependent Graphs

fmautner, emuchnik @ andrew.cmu.edu

May 5 2024

1 Summary

We set out to investigate the route planning problem in road networks. Road networks can be modelled as directed graphs where road segments for vertices and connections between roads form edges. The problem of assigning one car to a route is equivalent to the Single Source Shortest Path problem (SSSP), and the more general problem of planning routes for multiple cars can be interpreted as an All Shortest Paths variation (ASP), or a restricted set of sources in the Multiple Sources Shortest Paths variation (MSSP). This simple characterization, while easy to model, fails to accurately represent nuances of real road networks. As such, we have opted to use time-dependent dynamic graphs, where each edge in the graph has an associated time function T, mapping discrete time-steps $t \in \mathbb{Z}^+$ to real values $c \in \mathbb{R}$ describing the road's cost that time.

Our goal is to perform route planning for multiple vehicles on the same road network, taking into account their effects on the properties of the graph.

2 Background

A directed time-dependent graph G is defined as a triple

$$G = (V, E, T)$$

Where V is the set of vertices, E is the set of edges, and $T : E \times \mathbb{Z}^+ \to \mathbb{R}$ is a real-valued function mapping edges and a time-stamp to its corresponding 'wait time'. This is useful to describe roads whose behavior changes over time. For example, a highway might be empty and hence have low associated cost during early morning, but a high wait time during rush hour, and varying costs throughout the day. We could model such behavior by defining

$$T(e,t) = \begin{cases} 0 \text{ if } 00{:}00 \leq t < 8{:}00 \\ 5 \text{ if } 8{:}00 \leq t < 16{:}00 \\ 10 \text{ if } 16{:}00 \leq t < 20{:}00 \\ 2 \text{ if } 20{:}00 \leq t < 24{:}00 \end{cases}$$

As such, the associated weight with edge e changes depending on the time of day to appropriately model the road's occupancy.

2.1 Assumptions made

To bettwe model our problem, we made the following assumptions:

1. **First-In-First-Out (FIFO) property**[1]. This states that if a car x leaves a node u to node v over an edge $e = \{u, v\}$ at time t_x, and a car y does the same at time t_y, with $t_x \leq t_y$, then car y cannot arrive at v any earlier than car x. More formally,

$$t_x \leq t_y \implies T(e, t_x) + t_x \leq T(e, t_y) + t_y$$

   This is a realistic assumption in road networks, where we assume that road speeds are constant for all vehicles on it. This property is also important to allow for the problem of solving for shortest paths to be completed in polynomial time [1].

2. **Cars cannot wait at vertices.** This is a direct consequence from the FIFO property. It means that if a car arrives at vertex u at time t, it must also depart from it at time t, as for any node u, time t and edge $e = \{u, v\}$,

$$t + T(e, u) < t + \epsilon + T(e, t + \epsilon), \quad \forall \epsilon > 0$$

   We can model the real world as such by ignoring the fine-grained nature of traffic stops, instead amortizing them into the cost function T for the given edge.

2.2 Algorithms

Many popular graph algorithms have time-dependent extensions. In our work, we use an adapted version of Dijkstra's path finding algorithm for time-dependent graphs. The main difference is in how edge distances are relaxed, taking into account the current time and the time-dependent cost of a given edge. The initial, sequential algorithm is described below:

Algorithm 1 Single-Source Time-Dependent Dijkstra's (SS-TDD)

```
**Input:** Graph $G = (V, E, T)$, start time $t_s$, source $u_s \in V$
$N \leftarrow |V|$
Visited $\leftarrow$ [false] * $N$
minTime $\leftarrow$ [$\infty$] * $N$
minTime[$u_s$] $\leftarrow t_s$
$PQ \leftarrow [\{t_s, u\}]$
**while** $|PQ| > 0$ **do**
    $t_{curr}, u \leftarrow PQ$.pop()
    **if** not Visited[$u$] **then**
        **relax** ($u$, $t_{curr}$, $\{\{u', v'\} \in E | u' = u\}$)
    **end if**
**end while**
```

Algorithm 2 Edge relaxation procedure for SS-TDD

```
**Input:** Vertex $u$, current time $t_{curr}$, outgoing edges of $u$ $N(u) = \{\{u', v'\} \in E | u' = u\}$
**for** $e = \{u, v\}$ in $N(u)$ **do**
    arrival time $\leftarrow t_{curr} + T(e, t_{curr})$
    **if** arrival time < minTime[$u$] **then**
        minTime[$u$] $\leftarrow$ arrival time
        $PQ$.push(arrival time, $u$)
    **end if**
**end for**
```

These can also be extended into the Multiple Sources Shortest Path variation, or MS-TDD. The most obvious opportunity for parallelization in this algorithm is in the edge relaxation procedure. Since each edge goes to a distinct vertex (we assume no double edges), each can be processed independently and hence concurrently. This is where we start our optimization process to efficiently solve the routing problem.

3 Approach

We opted to write our solution in C++ using OpenMP, and ran our software on the GHC cluster machines supporting the platform choice. The mapping of parallel edge relaxation to processor threads is quite natural: Since each edge can be processed concurrently, we map all outgoing edges evenly to the available cores. This is done very simply with OpenMP directives. However, as discussed in the later sections, this initial approach proved sub-optimal, and many design iterations were carried out to optimize performance.

3.1 Problem representation and practical considerations

Due to the inherent sparsity of road networks, we opted for an adjacency list graph representation. We also modelled the time function from edges randomly, as we could not find such information available. To do so, each edge weight function is defined by two randomly sampled integers c and b, both in the range $[0, 10]$. The time function for an edge is then defined as

$$T(e,t) = (t \mod 25) * c + b$$

Note that this is a fairly arbitrary definition, but allows for simple graph representation and usage, while introducing the interesting complications brought by time-dependent edge weights. Another important note is the periodicity of the function, which more accurately models the capacity of roads over a 24 hour period. The numbers themselves $(0 - 10, 25)$ are arbitrary, chosen to create variable edge weights ranging from 0 to 250. These parameters can be easily customized to model specific networks differently.

To justify these choices, we describe the problem's representation and the performance debugging process that led us to this solution.

3.2 The Algorithm

In our final design, we use (for reasons described in the next sections) a sequential implementation of SS-TDD as a subroutine in a parallel routing algorithm for multiple cars over the road network. More formally,

Algorithm 3 Multiple Vehicle Routing

```
  **Input:** Vertices $\{u_1, u_2, \ldots, u_k\}$, current time $t_{curr}$, graph $G = (V, E, T)$
  **for** each *car* in parallel **do**
      run SS-TDD(graph, car)
      Every $N$ iterations, accumulate graphs
  **end for**
```

3.3 Performance debugging

In this subsection we outline the main adaptations we made along the way to the base SS-TDD implementation.

3.3.1 Naive parallelization over edges (edge relaxation)

This was our first approach at parallelizing the adapted Dijkstra's algorithm, and arguably the most natural thought to do so. However, this did not yield good results for multiple reasons. Firstly, it greatly undermines the available parallel processing power due to the inherent unpredictability of vertex degrees in road networks. The form of dynamic assignment used in this approach means that for all vertices with degree less than the number of processors, at least one must stay idle. Furthermore, the individual computation done by each thread is simple and quick, meaning that the bulk of the time spent by each thread is in synchronization and memory access. Another limiting factor to the efficacy of this strategy is the fact that most road networks are sparse graphs, meaning it is quite rare for a road segment (vertex) to have degree greater than the number of processors, further intensifying the issue previously listed. When testing empirically, the overhead of introducing parallelism was greater than the gained computation, and the sequential implementation was superior:

Table 1: Results from Chesapeake graph (see Section 4)

number of threads	relax + openMP	sequential	Speedup
1	0.0000193	0.0000275	1.4300000
2	0.0002512	0.0000275	0.0010954
4	0.0001465	0.0000275	0.1878224
8	0.0162781	0.0000275	0.0016906

The experiments were run 5 times and averaged. The only change between the sequential and parallel version is the inclusion of an OpenMP directive to parallelize the edge-relaxation loop, and of a critical section where the priority queue (which is not thread-safe) is updated. This was the bottleneck of the naïve parallel approach. It is also interesting to note the two order of magnitudes drop in the already poor performance between 4 and 8 threads. As less vertices can 'cover' the number of processors with out-edges, we see a sharp drop in performance, indicating the idleness of processors and synchronization costs.

3.3.2 Data Structure choices

We experimented with different graph and intermediary representations when implementing SSSP. One event where this happened was in the choice of using C++ sets to contain visited vertices on the graph, or boolean vectors that represent whether a vertex has been visited by our algorithm at a current iteration. Through empirical testing on the same algorithmic framework, we found the following:

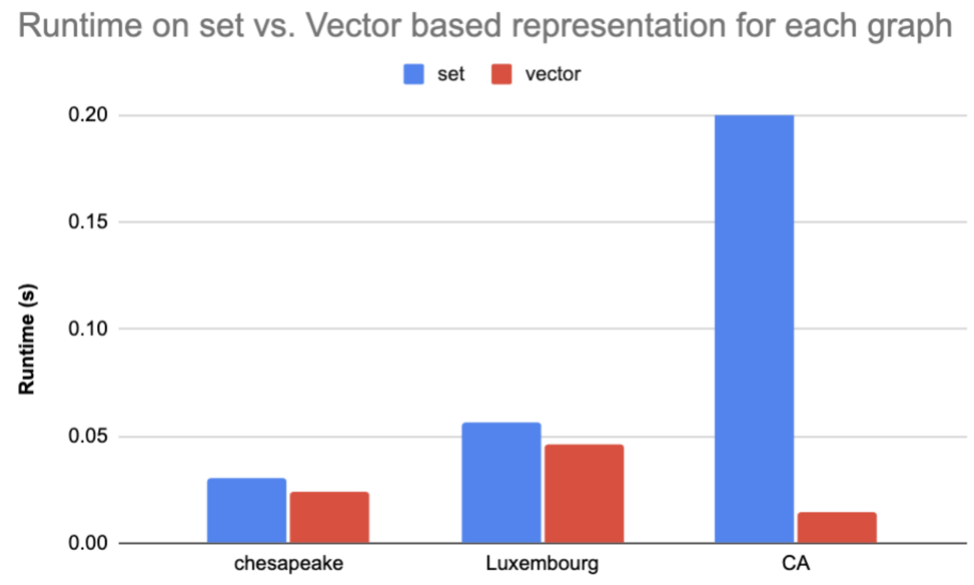


Figure 1: set vs. vector visited vertex representation

The experiments were run 5 times and averaged, the only change being the representation of visited vertices. We found that vector-based representation consistently outperformed set-based representation, and thus we used it in our final algorithm.

4 Experiments

This section describes the experiments carried out with the many versions of our approach. We used several small graphs to assist debugging, as well as three graphs containing real-world road data of varying size to test our implementation. They are:

- Chesapeake, VA [3]: This small graph contains 39 vertices and 170 edges that describe the principal road segments of this small city.

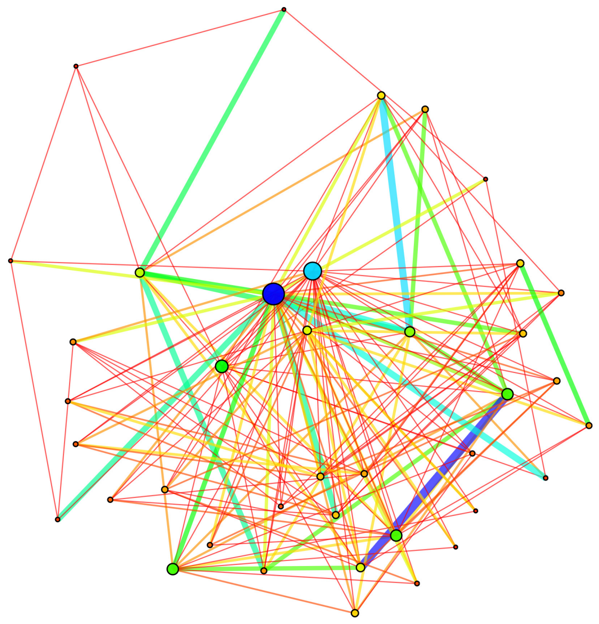


Figure 2: Visual representation of Chesapeake road graph

- Luxembourg [3]: This graph represents the road structure of the country of Luxembourg. It has 114599 vertices representing road segments and 119666 edges connecting them.
- California [2]: This large graph represents the road structure of the state of California. It has 1965206 vertices and 2766607 edges.

Some small adjustments were made to these files in order to match the required format and add time dependent weights as described in section 3.1. We measured the effect of parallelism in the multiple source variation of the problem.

We also analyze memory factors such as cache utilization and missing to investigate scalability concerns on individual SS-TDD implementations.

5 Results

We find a satisfactory near linear speedup in the route assignment task for multiple vehicles. The following shows a runtime and speedup chart for the Chesapeake road graph:

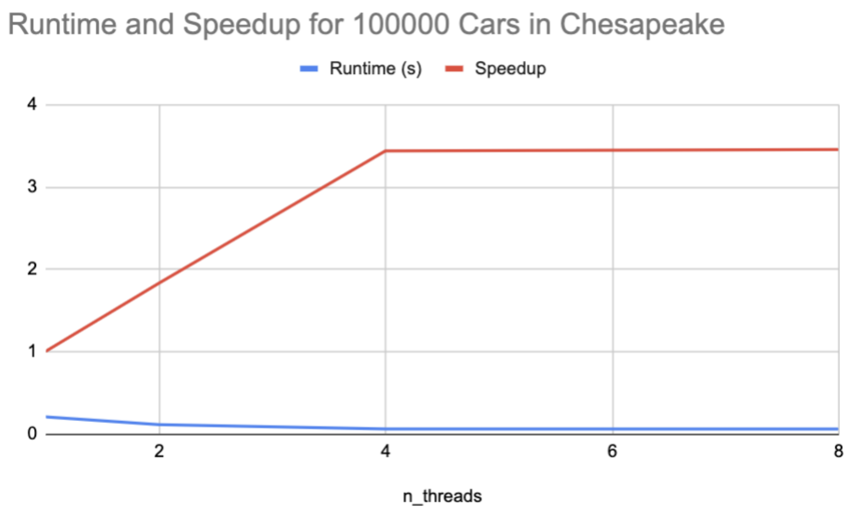
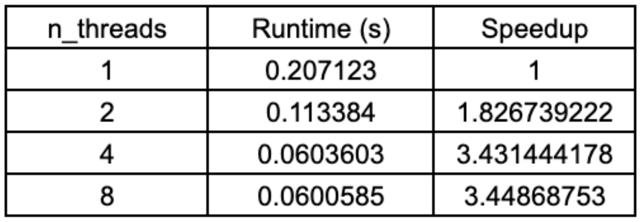


Figure 3: Speedup for Multiple Vehicle Routing



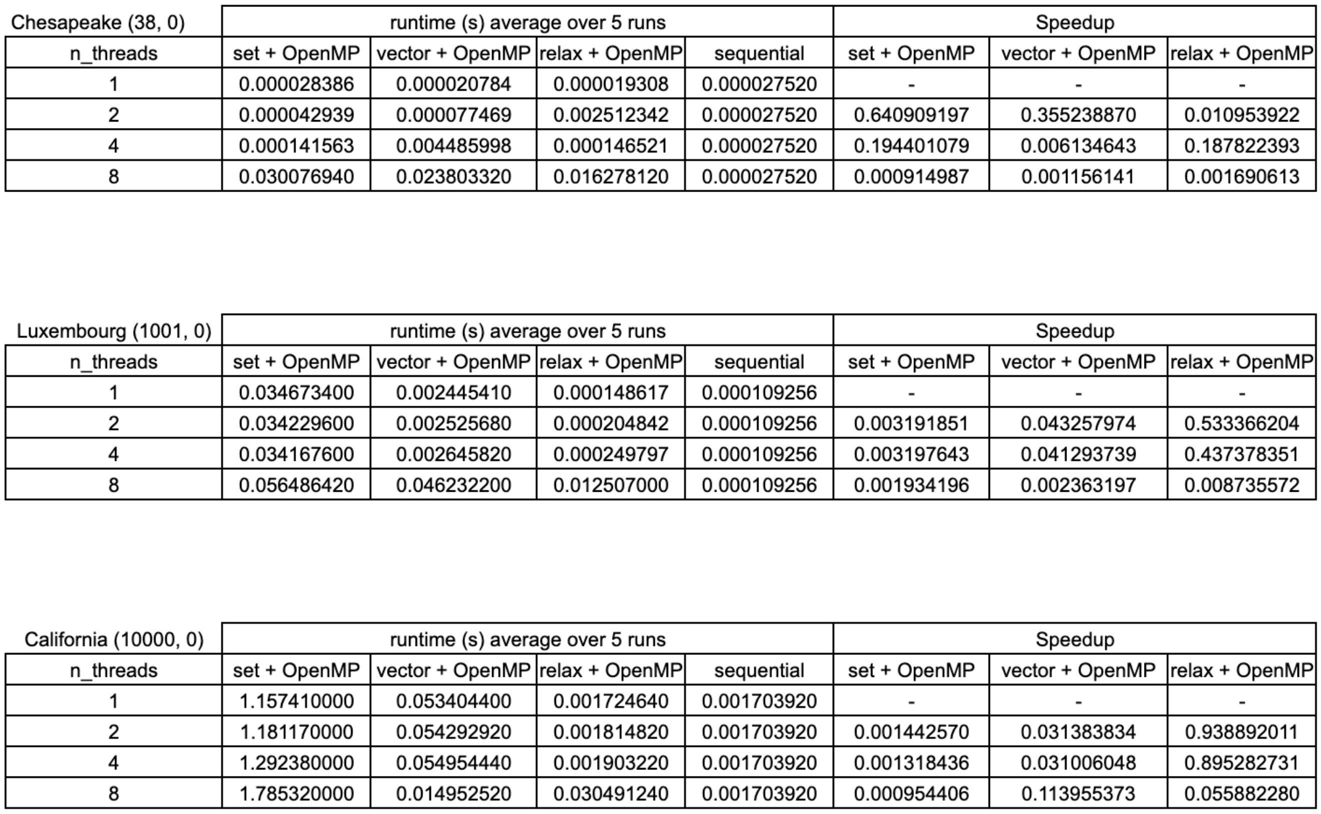
n_threads	Runtime (s)	Speedup
1	0.207123	1
2	0.113384	1.826739222
4	0.0603603	3.431444178
8	0.0600585	3.44868753

Figure 4: Speedup data for Chesapeake

We observe a near-linear speedup between 1 and 4 threads. This is a very reassuring result that indicates good use of parallel potential. However, there is no gained performance between 4 and 8 threads. The main reason for the near-linear speedups is the very limited synchronization steps, which we opted to in order to achieve good performance. However, this comes at the cost of route assignment quality, which is slightly inferior to the sequential algorithm. This however, is a tradeoff that we believe to be beneficial. To improve results, we need to adjust the parameter N, i.e. the number of iterations between synchronization. The current setting is 100.

5.1 Individual SS-TDD performance

The following tables contain information on all tested configurations of algorithmic and data structure choices. Each value is an average of 5 runs. The tuples before each table indicate the source vertex and the start time, set to 0 for all cases. As is visible, None of the edge-parallel approaches were able to outperform a well implemented sequential version.



Chesapeake (38, 0)	runtime (s) average over 5 runs				Speedup		
n_threads	set + OpenMP	vector + OpenMP	relax + OpenMP	sequential	set + OpenMP	vector + OpenMP	relax + OpenMP
1	0.000028386	0.000020784	0.000019308	0.000027520	-	-	-
2	0.000042939	0.000077469	0.002512342	0.000027520	0.640909197	0.355238870	0.010953922
4	0.000141563	0.004485998	0.000146521	0.000027520	0.194401079	0.006134643	0.187822393
8	0.030076940	0.023803320	0.016278120	0.000027520	0.000914987	0.001156141	0.001690613

Luxembourg (1001, 0)	runtime (s) average over 5 runs				Speedup		
n_threads	set + OpenMP	vector + OpenMP	relax + OpenMP	sequential	set + OpenMP	vector + OpenMP	relax + OpenMP
1	0.034673400	0.002445410	0.000148617	0.000109256	-	-	-
2	0.034229600	0.002525680	0.000204842	0.000109256	0.003191851	0.043257974	0.533366204
4	0.034167600	0.002645820	0.000249797	0.000109256	0.003197643	0.041293739	0.437378351
8	0.056486420	0.046232200	0.012507000	0.000109256	0.001934196	0.002363197	0.008735572

California (10000, 0)	runtime (s) average over 5 runs				Speedup		
n_threads	set + OpenMP	vector + OpenMP	relax + OpenMP	sequential	set + OpenMP	vector + OpenMP	relax + OpenMP
1	1.157410000	0.053404400	0.001724640	0.001703920	-	-	-
2	1.181170000	0.054292920	0.001814820	0.001703920	0.001442570	0.031383834	0.938892011
4	1.292380000	0.054954440	0.001903220	0.001703920	0.001318436	0.031006048	0.895282731
8	1.785320000	0.014952520	0.030491240	0.001703920	0.000954406	0.113955373	0.055882280

Figure 5: Condensed collected data on SSSP variations

5.2 Cache Misses and Scalability

Below we report the number of cache misses and per thread cache misses for the sequential and each of the set/ vector implementations of the parallel versions of TD-SSSP algorithms. These were collected using `perf` profiling.

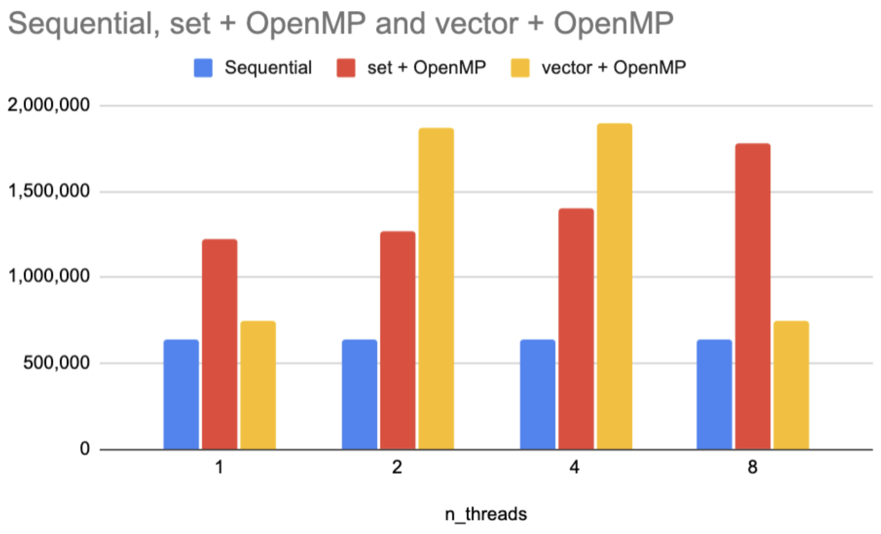


Figure 6: Total Cache Misses

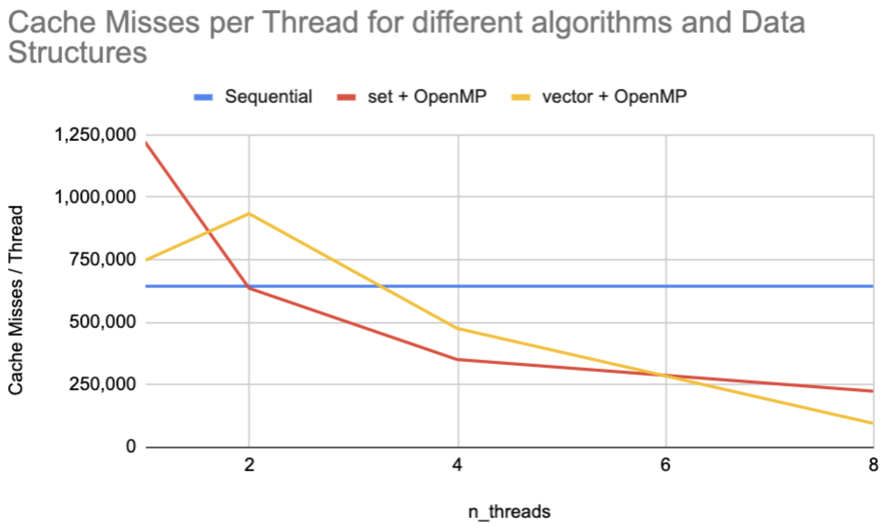


Figure 7: Cache Misses per Thread

The Figures above indicate that both the set and vector representation approaches indicate reasonably good scaling properties, with the number of cache-misses per thread decreasing about exponentially, at the same rate as the number of threads increases. However, we must note that the absolute number of cache misses in both approaches is consistently higher, at every thread count, than that of the sequential version. This likely happens due to false sharing when vertices are being processed. The high number of cache misses could also be seen as another reason for the vastly sub-optimal behavior of the parallel versions, although this is still more likely due to overhead and poor task allocation.

5.3 Limiting factors of performance

The most apparent limiting factor of parallel performance int he SSSP task is the sparsity of road network graphs. In the graphs used, very few vertices, if any, have more outgoing edges than there are processors available. This means that at anay given point in time, many processors may be idle. As discussed in 3.3.1, this also leads to poor load-balancing, and as a consequence the overheads of parallelism become harder to justify, especially since each processor is doing work that is quite low arithmetically dense. This means that processors spend disproportionately large portions of their time either idle or incurring synchronization costs.

5.4 Limited information based on graph size

Due to our underlying implementation, the size of the graph doesn't matter as much as how connected it is, in other words, the average degree of each vertex. This can be seen in examples where execution in the smaller Luxembourg graph was more costly than in the much larger California graph. This is sensible, as our version of TDD iterates only over those edges in the out-neighbor set of a vertex. If a graph is more sparse, it is more likely that it contains more disconnected components, and hence decrease total runtime. As such, an alternative investigation of the algorithms studied could involve graphs with constant number of vertices and varying degrees of connectiveness. This would likely yield more interesting and notable performance-related results, but would not be an accurate representation of road networks, which are sparse by nature.

5.5 Machine-related points of interest

All of our results were collected on GHC machines, more specifically, `ghc61`. These are multi-core machines. The same strategies employed could also be applied to both different hardware and implemented through different philosophies, such as GPU programming or message passing. It is important to reiterate, however, the inherent limitations of parallelism on sparse graphs.

6 Conclusion

The task of parallelizing shortest path algorithms over time-dependent graphs can be non-trivial based on graph specifications. For road networks, where the graphs are sparse, we found that parallelizing edge-related operations is inefficient, as rarely can a vertex saturate all processors with its edges. However, we found ample opportunity for parallelism in the Multiple Source Shortest Path problem, by using a fast sequential implementation of SS-TDD as a subroutine.

7 Next Steps

The primary area for improvement in our implementation lies in enhancing parallelism within the Single Source Shortest Path (SSSP) phase. Current strategies, such as parallelizing over vertices rather than edges, could significantly improve computational efficiency in sparse graphs. Approaches such as graph partitioning [4] and contraction hierarchies [1] provide promising avenues for achieving this. However, constructing a general, simple algorithm for time-dependent graphs remains a challenging task. Future work should also explore adaptive techniques that dynamically adjust the level of parallelism based on the graph's topology and the time-dependent characteristics of the edges. Such adaptivity could optimize performance across a variety of scenarios and help in scaling to even larger graphs. Finally, well tailored implementations of different approaches are generally best practice when it comes to extracting the most out of specific resources with specific constraints.

8 Work split

Felipe - 50%
Ethan - 50%

References

[1] Daniel Delling and Dorothea Wagner. "Time-Dependent Route Planning*". In: (). URL: `https://i11www.iti.kit.edu/extra/publications/dw-tdrp-09.pdf`.

[2] Jure Leskovec et al. *Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters*. 2008. arXiv: `0810.1355 [cs.DS]`.

[3] Ryan A. Rossi and Nesreen K. Ahmed. "The Network Data Repository with Interactive Graph Analytics and Visualization". In: *AAAI*. 2015. URL: `https://networkrepository.com`.

[4] Yuxin Tang, Yunquan Zhang, and Hu Chen. "A Parallel Shortest Path Algorithm Based on Graph-Partitioning and Iterative Correcting". In: *2008 10th IEEE International Conference on High Performance Computing and Communications*. 2008, pp. 155–161. DOI: `10.1109/HPCC.2008.113`.