CSE305 FINAL PROJECT REPORT

Marcel Chwiałkowski, Anca Sfia, Bruno Iorio

`https://github.com/bruno-iorio/Concurrent-Project`

In this project we explore the Δ-stepping algorithm for solving the single-source shortest path (SSSP) problem in graphs. The report firstly describes the basic Δ-stepping algorithm and its variations (static and dynamic implementations), and presents certain established results on its performance. Later, we take a look at the structure of our project – the algorithms that we implemented, helper functions for generating random graphs and scripts for analyzing results. Finally, we present the results of our experiments on the performance of different variants of Δ-stepping on different graphs and parameters, and compare it to regular serial Dijkstra.

Contributions to the project were distributed among the team members as follows:

1. Marcel Chwiałkowski – Dijkstra, serial Δ-Stepping, parallel Δ-stepping (static variant)
2. Anca Sfia – Serial and parallel Δ-stepping (variant with dynamic Δ)
3. Bruno Iorio – Benchmarking

The full source code is on the github:

- Static variant: `https://github.com/bruno-iorio/Concurrent-Project/blob/main/delta_step_static.cpp`
- Dynamic variant: `https://github.com/bruno-iorio/Concurrent-Project/blob/main/delta_step_dynamic.cpp`

1 Introduction

The Δ-stepping algorithm is an easily parallelizable solution to the SSSP problem, introduced in [2]. It principle, it works similarly to Dijkstra's algorithm - both algorithms maintain an array `dist` of tentative distances, which keeps the current distance of each vertex from the source throughout the runtime of the algorithm. At initialisation, each distance is set to $+\infty$, and throughout the runtime, as the algorithms scan through edges, new shorter paths are found and the tentative distances relax, finally yielding an array of exact shortest distances between vertices and the source. However, Dijkstra's algorithm goes vertex-by-vertex: it keeps a priority queue of unvisited nodes sorted by the lowest tentative distance, and at each

iteration it pops the first node of the queue, and relaxes all the edges going out from it. This presents a challenge for parallelization, since subsequent relaxations depend on each other, which makes it difficult to perform several relaxations at once,

In comparison, Δ-stepping maintains a set of buckets $[B_1, B_2, \dots]$ such that the bucket B_i holds unvisited vertices with tentative distance between $[i \times \Delta, (i+1) \times \Delta)$. Then, the buckets are emptied sequentially: starting from the non-empty bucket with the smallest index, all the edges coming out of vertices from this bucket are relaxed until the bucket is empty. This involves dividing the edges into heavy and light edges so that each light edge has weight less than Δ, and each heavy edge has weight of Δ or more. For a bucket B_i, firstly, all the light edges are relaxed at once - but since they have a weight of less than Δ, this means that some vertices can be inserted into B_i, thus this step is reiterated until B_i is empty. Afterwards, all heavy edges are relaxed at once - since their weight is more than Δ, the relaxed vertices will only be inserted to buckets with index larger than i.

This phase is repeated, at each time emptying the non-empty bucket with the smallest index - this smallest index grows at each step, as it is impossible for example when processing B_i, to insert a vertex into B_{i-1}. The algorithm eventually terminates when all the buckets are empty.

Pseudocode for the Δ-stepping algorithm (taken from [1]) can be seen below. Each bucket emptying phase consists of three loops: `Loop 1` empties the current bucket, while generating requests for relaxations of heavy and light edges (that is, saving to corresponding buffers, that a vertex should be updated to a different tentative dstance). `Loop 2` reads the relaxation requests for light edges, and performs them. Both 1 and 2 are locked in a while loop, as relaxations to light edges can result in reinsertions to the current bucket, so the while loop ensures that in the end the bucket is empty. Afterwards, `Loop 3` performs relaxation requests for heavy edges.

Algorithm 1 Δ-stepping SSSP

Require: $G(V, E)$, source s, bucket width Δ

for all $v \in V$ **do**
  define N_v^ℓ and N_v^h
  $d^\star(v) \leftarrow \infty, \quad B_\infty \leftarrow \{v\}$
end for
$B_\infty \leftarrow B_\infty \setminus \{s\},\ B_0 \leftarrow B_0 \cup \{s\},\ d^\star(s) \leftarrow 0$
$k \leftarrow 0$
while $k < \infty$ **do**
  $R_h \leftarrow \emptyset,\ R_\ell \leftarrow \emptyset$
  while $B_k \neq \emptyset$ **do** ▷ **Emptying buckets**
    for all $v \in B_k$ **do** (sequentially or in parallel) ▷ Loop 1
      $B_k \leftarrow B_k \setminus \{v\}$
      $R_\ell \leftarrow \textsc{GenRequests}(R_\ell, N_v^\ell)$
      $R_h \leftarrow \textsc{GenRequests}(R_h, N_v^h)$
    end for
    for all $(v, u, w) \in R_\ell$ **do** (sequentially or in parallel) ▷ Loop 2
      $R_\ell \leftarrow R_\ell \setminus \{(v, u, w)\}$
      $\textsc{Relax}(v, u, w)$
    end for
  end while
  for all $(v, u, w) \in R_h$ **do** (sequentially or in parallel) ▷ Loop 3
    $R_h \leftarrow R_h \setminus \{(v, u, w)\}$
    $\textsc{Relax}(v, u, w)$
  end for
  $k \leftarrow \min_i\{\, i : B_i \neq \emptyset \,\}$
end while
function $\textsc{GenRequests}(R, N_v)$
  for all $u \in N_v$ **do**
    $R \leftarrow R \cup \{(v, u, w(v, u))\}$
  end for
  return R
end function
function $\textsc{Relax}(v, u \in V,\ w \in \mathbb{R})$
  if $d^\star(v) + w < d^\star(u)$ **then**
    $i \leftarrow \left\lfloor \frac{d^\star(u)}{\Delta} \right\rfloor, \quad j \leftarrow \left\lfloor \frac{d^\star(v)+w}{\Delta} \right\rfloor$
    $B_i \leftarrow B_i \setminus \{u\},\ B_j \leftarrow B_j \cup \{u\}$
    $d^\star(u) \leftarrow d^\star(v) + w,\ p(u) \leftarrow v$
  end if
end function

Ensure: $d^\star(v),\ p(v) \quad \forall\, v \in V$

In contrast to Dijkstra, Δ-stepping is easily parallelizable - when emptying each bucket, the order of relaxation doesn't matter, so each loop can be executed by multiple threads.

1.1 Theoretical guarantees

[2] introduces a number of theoretical guarantees for the runtime of Δ-stepping in different settings. In this section we list some of these runtime bounds - however, it is worth noting that they depend on various difficult to estimate parameters. Thus, to judge the performance of this algorithm it is more useful to conduct experiments comparing it to Dijkstra.

The parameters of interest for bounding the runtime of Δ-stepping include:

- $n_\Delta = |C_\Delta|$: number of pairs of nodes connected by Δ-paths (simple paths with total weight not exceeding Δ).
- $m_\Delta = |C_{\Delta+}|$: number of triples (u, v, w) where u, v are connected by a Δ-path and (v, w) is a light edge.
- d: maximum degree.
- ℓ_Δ: maximum number of edges in a Δ-path plus one.
- L - maximum shortest path weight.

For a graph $G = (V, E)$ with $n = |V|, m = |E|$ and with arbitrary positive edge weights, sequential Δ-stepping runs in $O(n + m + n_\Delta + m_\Delta + \frac{L}{\Delta})$. The n_Δ and m_Δ factors bound the number of reinsertions (to the same bucket) and re-relaxations performed by the algorithm, and $\frac{L}{\Delta}$ bounds the number of buckets traversed. A parallelization of Δ-stepping in the CRCW PRAM model runs in $O(\frac{L}{\Delta} \cdot d \cdot l_\delta \cdot \log n)$ and needs work $O(n + m + n_\Delta + m_\Delta + \frac{L}{\Delta} \cdot d \cdot l_\Delta \cdot \log n)$ with high parallelism.

For graphs with random edge weights distributed uniformly over $[0, 1]$, the authors of [2] show that choosing $\Delta = O(\frac{1}{d})$ allows to limit the overhead caused by reinsertions and re-relaxations to $O(m + n)$. This allows to eliminate the n_Δ and m_Δ parameters from the complexity, resulting in a sequential runtime of $O(n + m + d \cdot L)$. The corresponding parallelization on CRCW PRAM runs in average in $O(d^2 \cdot L \cdot \log^2 n)$ with high parallelism.

More theoretical guarantees for different graphs and different variants of the algorithm are available in [2], however we considered them to be beyond the scope of this project.

Promises: The most important implication of these guarantees, is that with the overhead from reinsertions and re-relaxations limited to $O(n + m)$, sequential Δ-stepping achieves a similar complexity to Dijkstra, and parallel delta stepping theoretically has potential to be faster.

1.2 When does Δ-stepping excel?

Δ-stepping is especially effective when:

- The graph is large, so synchronization overhead is amortized.
- Edge weights have low variance, enabling large buckets and high parallelism.
- Graph degrees have a balanced distribution, so that each thread has a similar amount of work to process in each loop (this prevents other threads from waiting for one slow one to finish).
- The graph diameter is low, so few buckets must be processed and threads do not idle.

1.3 Challenges

1.3.1 Choosing the optimal Δ

Choosing the right value of Δ has a large impact on the performance of our algorithm. Notice that for graphs with integer weights, choosing $\Delta = 1$ effectively makes the Δ-stepping perform Dijkstra's algorithm - with no reinsertions to the same bucket, but no room for parallelization. Similarly, choosing a large Δ, makes the algorithm equivalent to Bellman-Ford - which is easily parallelizable, however it works in $O(mn)$. Therefore, Δ should be chosen to balance between available parallelism and number of reinsertions to one bucket. As [2] shows, in certain cases, choosing Δ proportional to $\frac{1}{d}$ is optimal. However in practice finding the right Δ is non-trivial.

1.3.2 Choosing the right number of threads

Choosing the number of threads close to our device's number of cores should be optimal - however, as we will see in the experiments, this is not always the case. For smaller graphs or graphs with skewed degree distributions (like road graphs), running more threads is less efficient, as each thread has less operations to process, or an imbalanced number of operations to process - and this overhead cancels out the benefits of parallelization.

2 Our implementations

In our project, we implement:

1. Two variants of parallel Δ-stepping: parallel Δ-stepping with static node partitioning and parallel Δ-stepping with static node partitioning and adaptive Δ (we call it dynamic Δ-stepping, however it is different from the dynamic node partitioning mentioned in [1] - a proper name for our implementation would be parallel Δ-stepping with static node partitioning and dynamic Δ).

2. serial Dijkstra for comparison.

2.1 Static Parallelization - Invariant Δ

This approach is taken from [1]. Upon initialization, all graph nodes are assigned randomly to threads. At each execution of `loop 1`, `loop 2`, `loop 3`, each thread processes only these vertices which are assigned to it - and the lists of requests are stored in 2-dimensional buffers, such that the requests at $R[fromID][toID]$ are written by thread $fromID$ in `Loop 1`, and read by thread $toID$ in `Loop 2` and `Loop 3`. This configuration avoids any race conditions - firstly, writing and reading are never occuring at the same time, and secondly when writing, each thread accesses the R buffer at a different $fromID$ and when reading, each accesses the R buffer at a different $toID$. This allows us to implement Δ-stepping lock-free.

2.2 Dynamic Parallelization – Adaptive Δ

The original Δ-stepping paper remarks that the "best" bucket width is not known prior to the runtime of the algorithm. In its conclusion, it recommends a novel approach of doubling Δ whenever the number of light-edge relaxation rounds inside the current bucket becomes excessive. [2].

Just as in the static parallelization with invariant delta, vertices are randomly assigned threads at initialisation and are grouped into buckets based on their tentative distances. While processing a bucket, the algorithm repeatedly relaxes its light edges. If this inner loop must iterate more than `delta_update` times, the width Δ is interpreted as too small. At that point the algorithm:

1. sets $\Delta \leftarrow 2\Delta$,
2. re-partitions every adjacency list into new light/heavy subsets,
3. rebuilds the bucket array and re-inserts all still-live vertices, and
4. clears and repopulates a global min-heap that tracks which bucket indices are non-empty.

The pseudocode for this operation is presented below, and it follows at the end of the pseudocode for the static version to build the dynamic version.

Algorithm 2 Rebuild Procedures for Dynamic Δ-Stepping

function CHECKREBUILD(*lightRounds*)
  if *lightRounds* $>$ *delta_update* **and** $\Delta < 2\Delta_{\max}$ **and** *rebuild_cnt* $<$ *MAX_REBUILDS* **then**
    REBUILDBUCKETS
  end if
end function
function REBUILDBUCKETS
  $\Delta \leftarrow 2\Delta$
  rebuild_cnt $\leftarrow$ *rebuild_cnt* $+ 1$
  for all $v \in V$ **do** (sequentially or in parallel) ▷ Re-partition neighbors
    $N_v^{\ell} \leftarrow \emptyset, \quad N_v^{h} \leftarrow \emptyset$
    for all $(v, u, w) \in E$ **do**
      if $w \leq \Delta$ **then**
        $N_v^{\ell} \leftarrow N_v^{\ell} \cup \{(u, w)\}$
      else
        $N_v^{h} \leftarrow N_v^{h} \cup \{(u, w)\}$
      end if
    end for
  end for
  live $\leftarrow \bigcup_i B_i$ ▷ Collect live vertices
  for all i **do**
    $B_i \leftarrow \emptyset$
  end for
  newSize $\leftarrow \lceil \frac{maxDist}{\Delta} \rceil$ ▷ New number of buckets
  RESIZEBUCKETS($\{B_i\}$, *newSize*)
  activeBuckets $\leftarrow \emptyset$
  for all $v \in$ *live* **do**
    $j \leftarrow \left\lfloor \frac{d^{\star}(v)}{\Delta} \right\rfloor$ ▷ Recompute bucket index
    $B_j \leftarrow B_j \cup \{v\}$
    if $j \notin$ *activeBuckets* **then**
      activeBuckets $\leftarrow$ *activeBuckets* $\cup \{j\}$ ▷ Update min-heap
    end if
  end for
end function

The min-heap stores the indices whose bucket lists are non-empty. Whenever the first vertex enters an empty bucket, its index is pushed. The main loop then pops the smallest index for the next phase. This avoids a linear scan over the bucket array and has cost $O(\log |H|)$ per update (where H is the number of non-empty buckets currently tracked on the heap), which is negligible compared to relaxation work. A similar priority queue for active buckets is suggested in the implementation notes of the original paper. [2]

Only vertices whose tentative distance has not improved since their last insertion are moved during a rebuild, so all classical Δ-stepping invariants are preserved. Each rebuild

takes $O(|V| + |E|)$ in the worst case, but the cost remains negligible for sparse graphs also due to the low *MAX_REBUILDS* (default 5) cap.

Conceptually, this adaptive variant removes some of the tuning sensitivity of classical Δ-stepping while incurring an overhead on graphs where the initial Δ was already near-optimal.

2.3 Design choices

Here we describe and justify some of the non-obvious design choices that we made when implementing the algorithms:

1. The requests buffers are defined as:
   `std::vector<std::vector<std::vector<std::pair<int, double>>> ReqLight, ReqHeavy;`
   Thus, each list of requests passed from thread A to thread B is a `std::vector` - as we only need to iterate through it in any order, we don't need a different data structure.

2. For the static algorithm, buckets are defined as:
   `std::vector<std::vector<std::unordered_set<int>> buckets;` where each bucket is distributed between the threads. The portions of the buckets belonging to each thread are implemented as `std::unordered_set`, as we need $O(1)$ insertion and deletion (not necessarily from the back!). In paper [2] this is implemented with doubly linked lists, however due to time constraints we opt for sets, which have the same theoretical complexity.

   For the dynamic algorithm, buckets are defined as:

   `std::vector<std::vector<std::vector<int>> buckets;` where each bucket is distributed between the threads. We store the per-thread contents of each bucket as a `std::vector<int>` for simplicity and fast iteration. Although deletions from the middle of a vector are not $O(1)$, the typical usage involves repeated insertions and scans, not random deletions. Vectors also offer good cache locality during parallel relaxations. The main reason for the change from the static version, however, is the fact that rebuilding unordered set buckets takes a longer time than rebuilding vector buckets.

3. The threads contain elements from `C++20`, namely `std::barrier`. We found it to be the easiest way to parallelize Δ-stepping: instead of recreating every thread at each execution of `Loop 1`, `Loop 2` and `Loop 3`, we create each thread at the beginning of the algorithm, and synchronize them at each phase with the barrier - for example, main cannot start `Loop 2` until each thread finishes `Loop 1`.

4. In the static version, checking which buckets are not empty is a simple linear scan - we use the fact that when a bucket with index i is emptied, it will never fill again, so we always start the scan from the last emptied index. This yields an amortized complexity of $O(L/\Delta)$ for the scan. In the dynamic version, to keep track of which buckets are not empty, we maintain a global min-heap of bucket indices:
   `std::priority_queue<int, std::vector<int>, std::greater<int>> activeHeap;`
   This min-heap allows the main loop to quickly retrieve the smallest non-empty bucket index without scanning the entire bucket array. This structure and usage follow suggestions in [2].

2.4 Graph formats

We decided to represent graphs as adjacency lists - this allowed us to process graphs with a large number of vertices (which would be impossible when using adjacency matrices). For loading graphs from files, we used the `.gr` format from `9th DIMACS Implementation Challenge - Shortest Paths` link here. This allowed us to test our implementation on the large road graphs provided on the DIMACS website. `.gr` files represent graphs in plain text as lists of edges. Each `.gr` file includes a line that starts with the characters `p sp` and specifies the number of vertices and edges, and a number of lines starting with `a`, each one describing a single edge. Lines starting with `c` are comments and are ignored by our graph parser. A good example to get familiar with this format is the `test.gr` file, found in the graphs folder linked to our report.

3 Benchmarking

3.1 Dataset

As part of our investigation, we benchmarked the performance of the delta stepping algorithm on a collection of road networks of large regions of the United States. These tests may provide us information of real applicabilities and limitations of choice of implementation of our algorithms. The dataset is available on the DIMACS website here.

3.2 Random Graph (RG) Generators

To generate random graphs, we have a function `wideWeightRandomGraph(n,m,w_min,w_max, G)`, which selects randomly assigns values between $[w_{\min}, w_{\max}]$ to m edges of the empty graph G, which has n vertices. To generate a random edge, we randomly choose 2 vertices. If they are the same vertices or if this edge was already assigned, we just drop it, otherwise, we assign a random value between $[w_{\min}, w_{\max}]$ to it.

3.3 Experimental Setup

For this project, we used Ecole Polytechnique computers to run our tests. Specifically, all tests were ran in the `ain.polytechnique.fr` login.

We perform the following tests for both road network graphs and *RG*s for our algorithms:

- tests over the values of delta
- tests over the number of threads
- tests over the density of the graph (only for *RG*)

For each evaluation, we compute an average over 5 runs of the algorithm. After performing all these tests, we will compare the performance of each implementation of our algorithm with Dijkstra's algorithm.

4 Results

Firstly, we will present the performance of each algorithm on the road network of the state of California. This network contains 1,890,815 nodes and 4,657,742 edges.

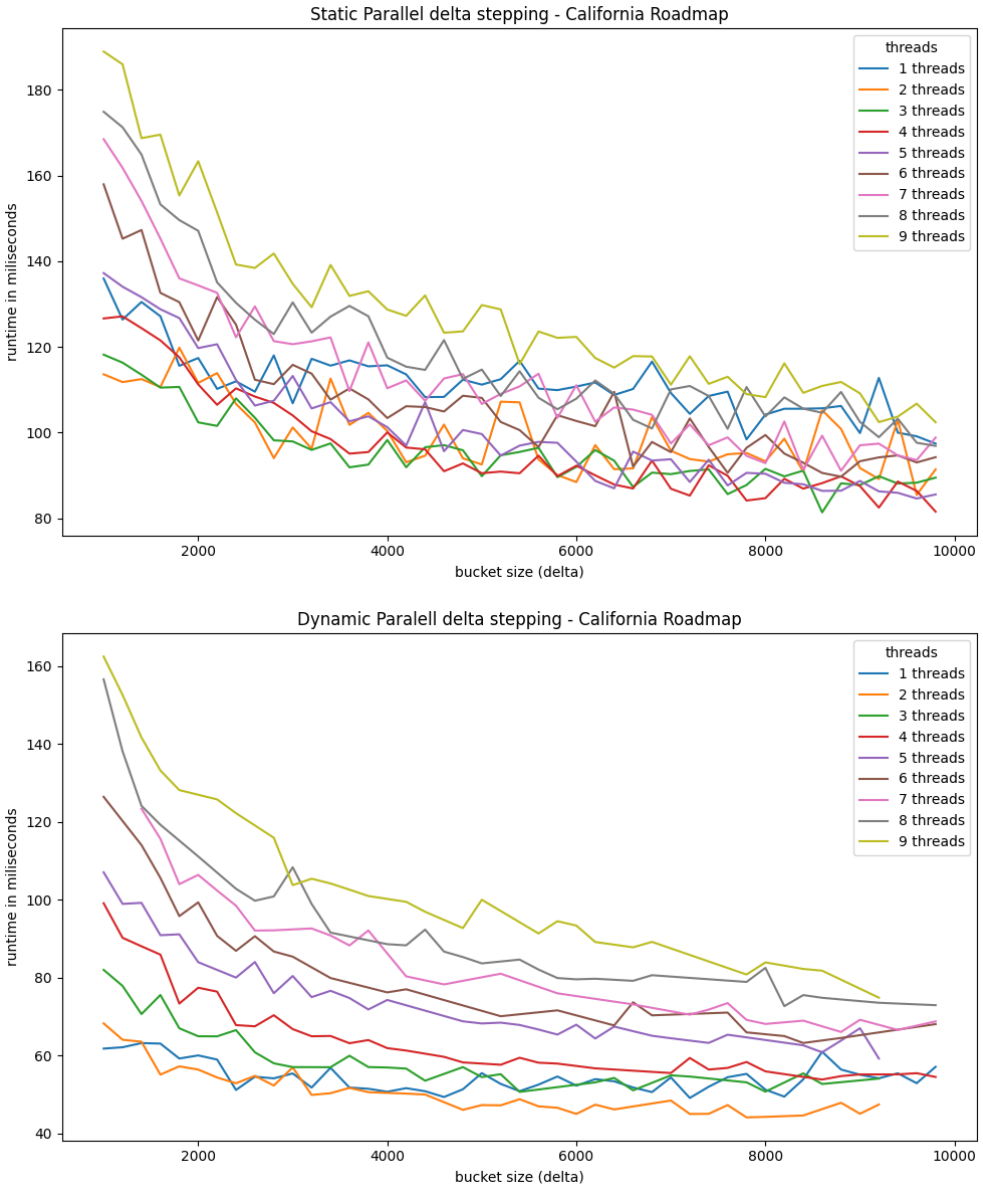


Figure 1: Delta stepping on Road network

The graphs above reveal a significant speedup for larger values of Δ. However, there isn't necessarily a speedup when we increase the number of threads. We can see that the best setup for the static variant happened when the number of threads was equal to 4, meanwhile, the dynamic variant performed better with 2 threads. This is because road graphs (like the one we ran the algorithms on), have skewed degree distributions and high variance of edge weights, which leads to uneven work distribution between threads.

We analyse the CPU load of the algorithms:

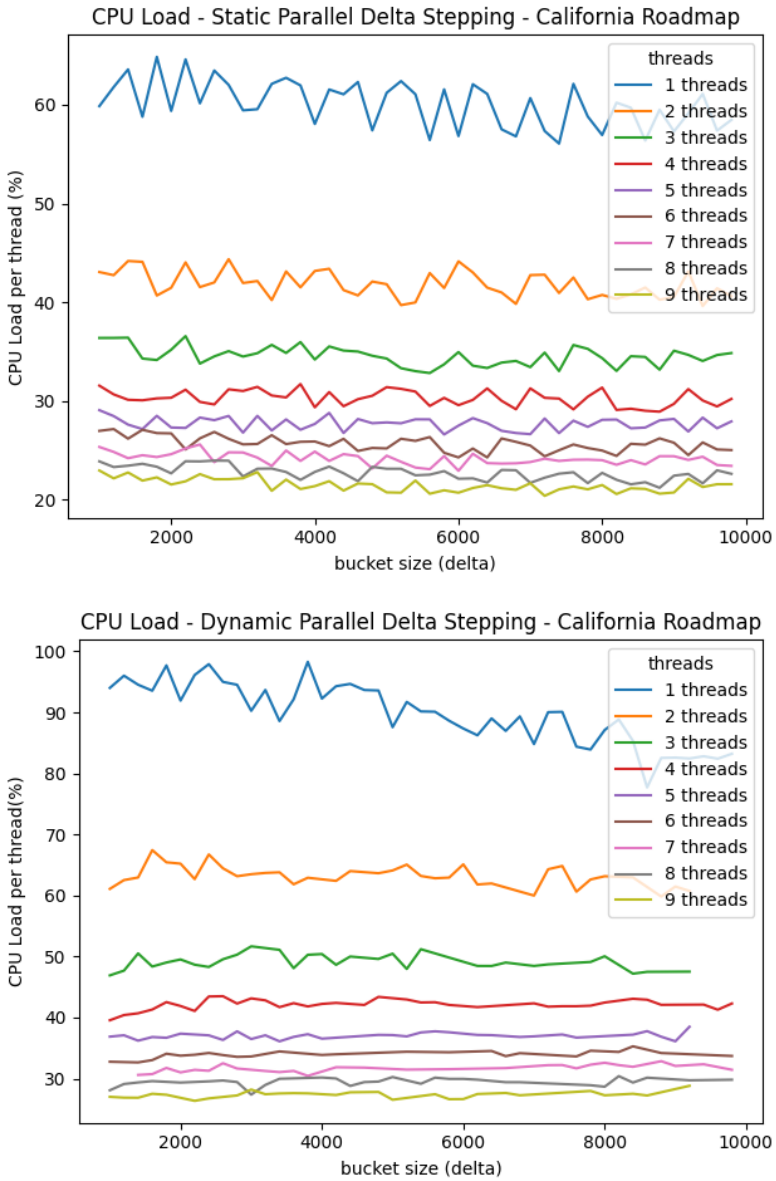
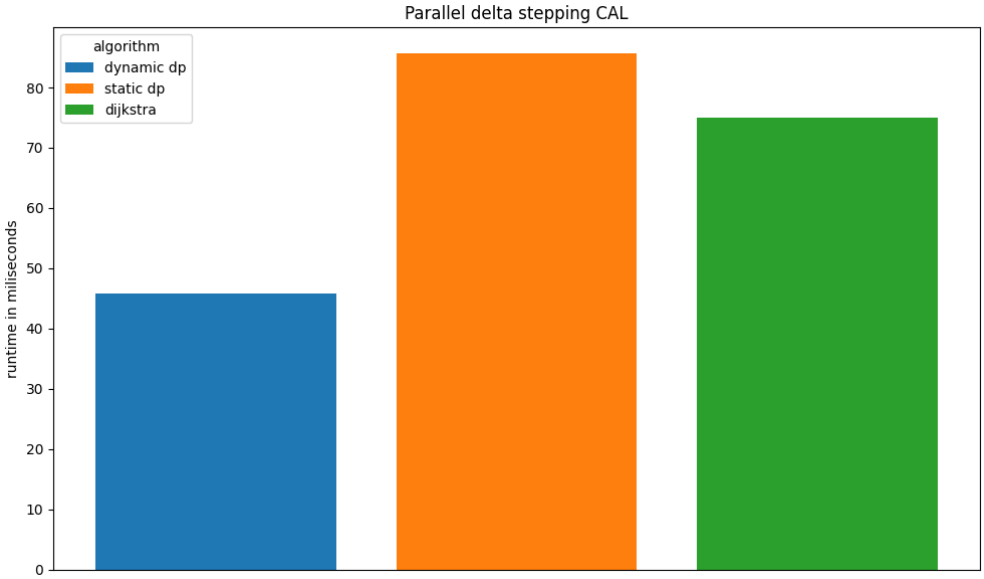


Figure 2: CPU load per thread

We notice that there is a significant decrease in CPU load per thread as we increase the number of threads. This is an indicator that the more threads we use, the algorithm waits significantly more(e.g. threads waiting the execution of other threads). This confirms our hypothesis from the previous paragraph, and is likely due to skewed degree distributions and high variance of edge weights in the California roadmap.

When we consider the static variant with parameters $\Delta = 9000$ and `num_threads` $= 4$, and the dynamic variant with parameters $\Delta = 8000$ and `num_threads` $= 2$, we get the following graph:



In this case, the static variant was 13% slower than Dijkstra, whereas the dynamic variant had a speed up of 163% when compared to Dijkstra.

Now, we consider random graphs $RG(n, c)$, where $n = 2^{17}$ and $c = 32$, where the weights of each edges are between 0 and 1. We expect Δ to be in the range $[0, 1]$. Below we compute the execution time of the dynamic variant under different values of Δ and `thread_num`:

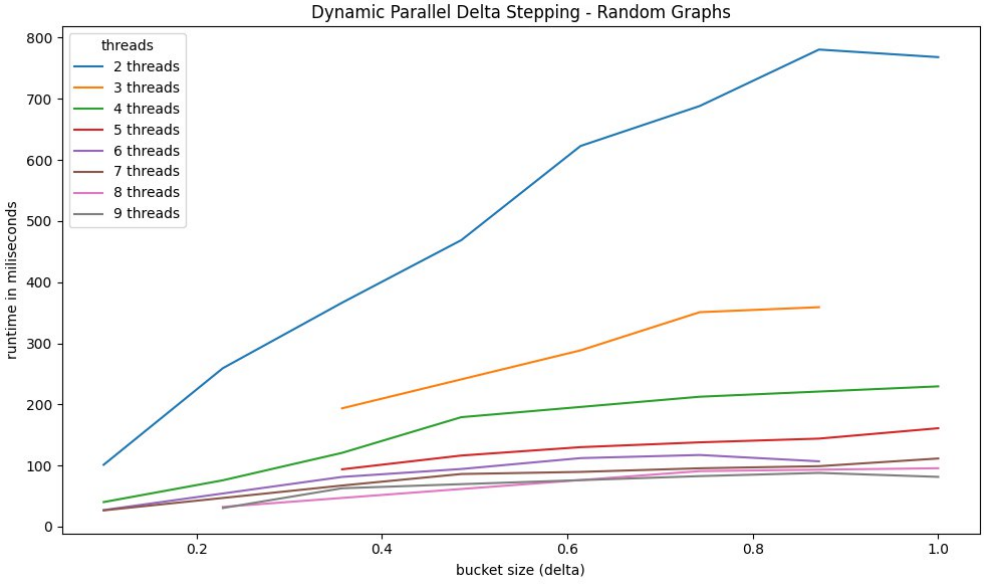


Figure 3: Delta stepping on Random graphs

In this case, using more threads actually positively affected the performance of the model in terms of speed. This is likely because random graphs have much more regular degree distributions and smaller edge weight variance.

Compared to the CPU load per thread on the road map graph, here a larger number of threads retains a higher CPU load per thread (for 9 threads, around 30%, compared to the previous 25%, keeping in mind that the roadmap of California has around 10 times more vertices).

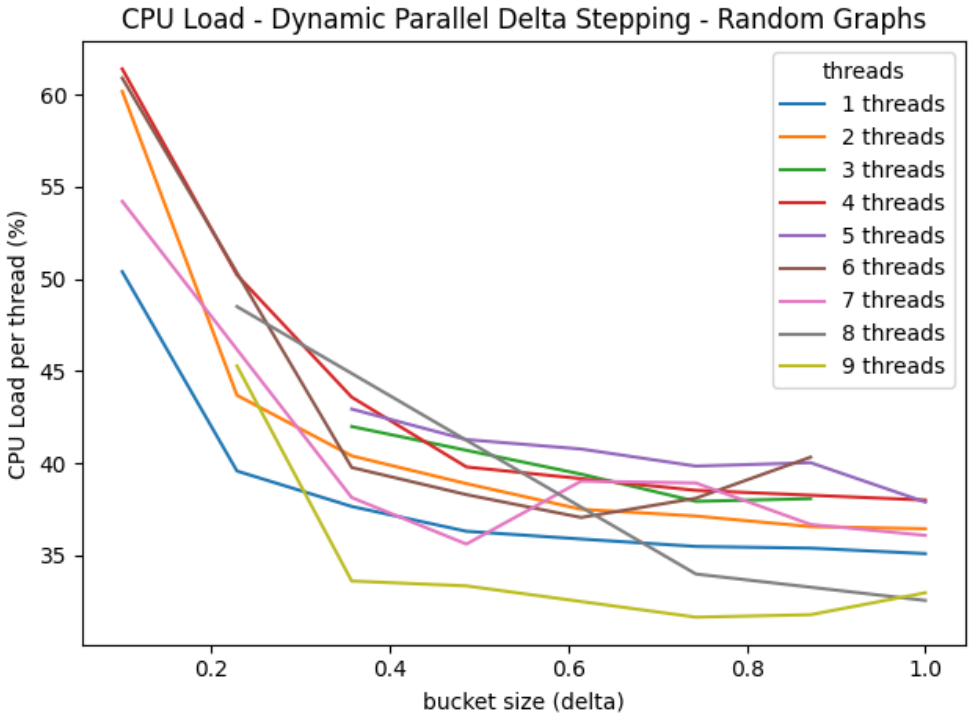


Figure 4: CPU load per thread

We finally tested these fixed parameters to find out the overall speedup we had for Random Graphs, Below, we vary the graph density to firstly compare the algorithms, and to check how the algorithm performs in dense and sparse graphs.

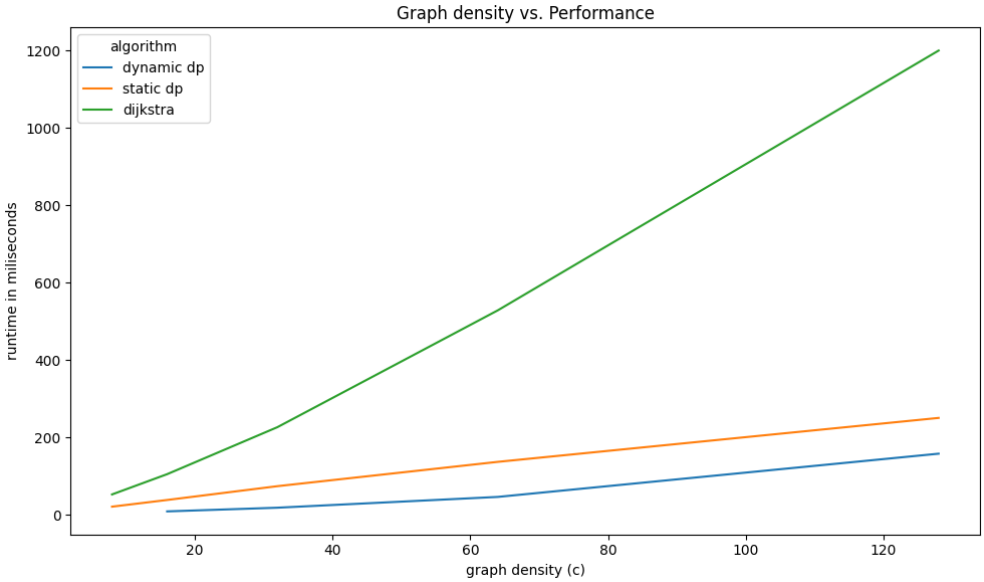


Figure 5: Fixed parameters on RG

We found a speedup of 1070% for the dynamic variant and 360% for the static variant when compared to Dijkstra.

5 Analysis of results

Our benchmarking revealed that for certain graphs, our algorithms experience a significant improvement in speed, competing with Dijkstra. On the dense random graphs, both dynamic and static outperform Dijkstra by at least 1 order of magnitude, and on larger road graph they have similar performance.

It is also important to note that the dynamic variant outperforms the static variant in our tests. This might be due to implementation differences: the static variant does a linear scan when finding a non-empty buckets and the dynamic variant keeps a min-heap of non-empty buckets, and both algorithms use different data structures for buckets.

References

[1] Duriakova, E., Ajwani, D., Hurley, N.: Engineering a parallel delta-stepping algorithm. In: 2019 IEEE International Conference on Big Data (Big Data). pp. 609–616 (2019). https://doi.org/10.1109/BigData47090.2019.9006237

[2] Meyer, U., Sanders, P.: Delta-stepping: a parallelizable shortest path algorithm. Journal of Algorithms **49**(1), 114–152 (2003). https://doi.org/https://doi.org/10.1016/S0196-6774(03)00076-2, `https://www.sciencedirect.com/science/article/pii/S0196677403000762`, 1998 European Symposium on Algorithms